

p. 68

Theoretical Foundations for Information Representation and Constraint Specification

**Christopher P. Menzel
Richard J. Mayer**

***Knowledge Based Systems Laboratory
Texas A&M University***

March 5, 1991

Cooperative Agreement NCC 9-16

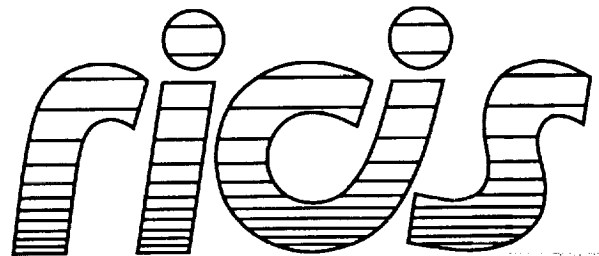
**Research Activity No. IM.06:
Methodologies for Integrated
Information Management Systems**

**NASA Johnson Space Center
Information Systems Directorate
Information Technology Division**

N92-25149

Unclas
0086865

(NASA-CR-190286) THEORETICAL FOUNDATIONS
FOR INFORMATION REPRESENTATION AND
CONSTRAINT SPECIFICATION (Research Inst.
for Computing and Information Systems) 68 p
CSCL 05B G3/62



*Research Institute for Computing and Information Systems
University of Houston-Clear Lake*

TECHNICAL REPORT

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information Systems (RICIS) in 1986 to encourage the NASA Johnson Space Center (JSC) and local industry to actively support research in the computing and information sciences. As part of this endeavor, UHCL proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a continuing cooperative agreement with UHCL beginning in May 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The UHCL/RICIS mission is to conduct, coordinate, and disseminate research and professional level education in computing and information systems to serve the needs of the government, industry, community and academia. RICIS combines resources of UHCL and its gateway affiliates to research and develop materials, prototypes and publications on topics of mutual interest to its sponsors and researchers. Within UHCL, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business and Public Administration, Education, Human Sciences and Humanities, and Natural and Applied Sciences. RICIS also collaborates with industry in a companion program. This program is focused on serving the research and advanced development needs of industry.

Moreover, UHCL established relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research. For example, UHCL has entered into a special partnership with Texas A&M University to help oversee RICIS research and education programs, while other research organizations are involved via the "gateway" concept.

A major role of RICIS then is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. RICIS, working jointly with its sponsors, advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research and integrates technical results into the goals of UHCL, NASA/JSC and industry.

***Theoretical Foundations
for
Information Representation
and
Constraint Specification***

1. The first part of the document is a list of the names of the persons who were present at the meeting.

2. The second part of the document is a list of the names of the persons who were absent from the meeting.

3. The third part of the document is a list of the names of the persons who were present at the meeting.

RICIS Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Dr. Christopher P. Menzel and Dr. Richard J. Mayer of Texas A&M University. Dr. Peter C. Bishop served as RICIS research coordinator.

Funding was provided by the Air Force Armstrong Laboratory, Logistics Research Division, Wright-Patterson Air Force Base via the Information Systems Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between the NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this research activity was Robert T. Savely of the Information Technology Division, NASA/JSC.

The views and conclusions contained in this report are those of the authors and should not be interpreted as representative of the official policies, either express or implied, of RICIS, NASA or the United States Government.

**Theoretical Foundations
for
Information Representation
and
Constraint Specification**

**Christopher Menzel and Richard J. Mayer
Knowledge Based Systems Laboratory
Texas A&M University**

March 5, 1991

Preface

This paper describes the research accomplished at the Knowledge Based Systems Laboratory of the Department of Industrial Engineering at Texas A&M University. Funding for the lab's research in Integrated Information System Development Methods and Tools has been provided by the Air Force Human Resources Laboratory, AFHRL/LRL, Wright-Patterson Air Force Base, Ohio 45433, under the technical direction of USAF Captain Michael K. Painter, under subcontract through the NASA RICIS Program at the University of Houston. The authors and the design team wish to acknowledge the technical insights and ideas provided by Captain Painter in the performance of this research as well as his assistance in the preparation of this report.

Summary

A method can be thought of as a distillation of good practice for a particular system development situation. Formalization of a successful engineering, management, production, or support technique into a method is done in hopes of raising the performance of the novice practitioner to a level comparable with that of an expert through the appropriate use of the method. Individual methods are normally accompanied by a special purpose graphical language that serves to provide focus and display emphasis for the major concepts that need discovery, consensus, or decision relative to a specific system development life cycle activity. Experience has proven that the personal and organizational preferences for particular methods are likely to make it necessary to isolate the information gathered and displayed by one method in such a way that it can be used in other stages of the life cycle or be displayed in alternative forms.

This paper outlines the theoretical foundations necessary to construct a Neutral Information Representation Scheme (NIRS) which will allow for automated data transfer and translation between model languages, procedural programming languages, database languages, transaction and process languages, and knowledge representation and reasoning control languages for information system specification.

Contents

Introduction	5
1 Motivation	5
2 First-order Languages	6
2.1 Vocabulary	7
2.2 Grammar	10
3 First-order Semantics	13
3.1 Structures and Interpretations	13
3.1.1 Interpretations of Constants and Function Symbols . .	13
3.1.2 Interpretations of Predicates	14
3.2 Truth	15
3.2.1 Variable Assignments	15
3.2.2 Truth Under an Assignment	16
3.2.3 Truth	19
4 Logic	19
4.1 Propositional Logic	19
4.1.1 Axioms for Propositional Connectives	20
4.1.2 Rules of Inference: Modus Ponens	21
4.2 Predicate Logic	22
4.2.1 Axioms for the Quantifiers	22
4.2.2 Rules of Inference: Generalization	24
4.3 Identity	25
4.3.1 Identity and Expressive Power	25
4.3.2 Axioms for Identity	26
5 Constraint Languages	28
5.1 Basic Set Theory	29
5.1.1 Membership	29
5.1.2 Basic Set Theoretic Axioms	30
5.1.3 Finitude and the Set of Natural Numbers	34
5.1.4 Difference, Intersection, and the Empty Set	34

5.1.5	Functions and Ordered n -tuples	35
5.1.6	The Intended Semantics: The Cumulative Hierarchy of Sets	36
5.2	Constraints Revisited	37
5.3	Information Structures: An Intuitive Account	38
6	Summary	41
	Appendix A - An Overview of IDEF1	43
	Appendix B - Formal Information Structures	56

Introduction

This document presents the theoretical foundations for information representation languages of both graphical and textual varieties. It is intended to serve as a framework for providing rigorous syntax and semantics of existing and proposed information analysis, design, and engineering methods. The purpose of such a framework is to provide information representation language designers with the guidance necessary to allow for automated inter-model data transfer and translation. Thus, this document should be viewed as the structure for an information model data exchange specification. Finally this theory is motivated by the need for a general theory of information representation. Thus, this theory serves as the first step towards achievement of a Neutral Information Representation Scheme (NIRS) for an Integrated Development Support Environment (IDSE) that can serve as the platform for a seamless Computer Aided Software Engineering (CASE) environment. Section 1 of this document describes the motivations and considerations behind the proposed theory. Section 2 introduces a restricted first-order language syntax that is proposed as the bounding syntactic structure for information modeling languages. Section 3 provides a model theoretic semantics for those languages, and Section 4 a corresponding logic. Section 5 describes the application of these concepts to constraint languages.

1 Motivation

The Air Force Integrated Information Systems Evolution Environment (IISEE) project represents a comprehensive research effort to develop technologies critical to effectively manage, control, and exploit information as a resource. The resulting developments will provide integration support methodologies, frameworks, and experimental tools to support integrated information management systems development and evolution.

One of the key premises on which this program is based is the recognition of the need for a suite of information modeling methods to service the large number of tasks and user/developer roles in an evolutionary integrated information system development process. Each method in this suite is de-

signed to serve a particular class of human users performing specific tasks or decision processes. The individual methods normally are accompanied by a special purpose graphical language that serves to provide focus and display emphasis for the major concepts that need discovery, consensus, or decision relative to that task. The problem with this approach is that these syntactic features restrict the information that can be stated in the language.

The seamless CASE concept is focused on development of the technological components and management methods for seamless software engineering environments. The term "seamless" is meant to convey the integrated nature of the methods and tools provided to the software implementer. The pluralization of the term "environments" is meant to convey the fact that different seamless case environments will be defined for different software types.

This particular document is the result of research which began as an effort to define a constraint specification language for a particular information modeling method known as IDEF1.¹ An overview of the method and its formalization are found in Appendices A and B. As the effort progressed, it was recognized that the emerging language structures were similar to those being investigated for the conceptual schema representation language for the IDSE seamless CASE environment and for the Neutral Information Representation Scheme to be used to provide the basis for an evolving system description capable of supporting automated knowledge based model translation. The theory presented in this report has been used as the formal foundation for a family of languages that will serve the above described purposes. This family of Information System constraint languages (ISyCL) is described in [1].

2 First-order Languages

The basis of our account will be the notion of a *first-order language*. First-order languages are flexible, expressively quite rich, and extremely well understood. They are used extensively in mathematics, linguistics, philosophy,

¹See, e.g., [2] and [3]. "IDEF" was originally an acronym for "ICAM Definition Language," but the suite of IDEF methods has since evolved independently of its ICAM origins. Hence, like "NCR" (formerly an acronym for "National Cash Register"), "IDEF" is now simply a name like "George," and an acronym no longer.

and computer science whenever clarity of expression is especially important. Many familiar mathematical theories such as the theory of sets, boolean algebra, topology, etc., can be elegantly expressed in first-order terms. More recently first-order languages have found their way into the domain of artificial intelligence, where first-order languages find straightforward representation in familiar AI programming languages like LISP and PROLOG. Indeed, first-order mathematical logic is the formal foundation of PROLOG—an acronym for PROgramming in LOGic.²)

Generally speaking, a first-order language \mathcal{L} is a *formal* language. That is, it is a formal structure consisting of a fixed set of basic symbols, often called the *vocabulary* of \mathcal{L} , and a precise set of syntactic rules, its *grammar*, for building up the proper sentences, or *formulas*, of the language that are capable of bearing information.

2.1 Vocabulary

The basic vocabulary of a first-order language consists of several kinds of symbols:

- Constants
- Variables
- Function symbols
- Predicates
- Logical symbols.

Constants are symbols that correspond to names in ordinary language. For many purposes, it is useful to use abbreviations of names straight out of ordinary language for constants, e.g., j for John, wp for Wright-Patterson, v for Venus, o for Ohio, etc. When we are describing languages in general and have no specific application in mind, we will simply use the letters a , b , c , and d , perhaps with subscripts; we will assume that we will add no more than

²See, e.g., [4].

finitely many subscripted constants to our language.³ Constants are usually lower case letters, with or without subscripts, but this is not necessary. Indeed, it is often useful to use upper case.

We will often want to say things about an "arbitrary" constant as a way of talking about all constants, much as one might talk about an arbitrary triangle ABC in geometry as a way of proving something about all triangles in general. For this purpose it will not do to talk specifically about a given constant, a say, since we want what we say to apply to *all* constants generally. This requires that, when we are talking *about* our language, we use special *metavariables* whose roles are to serve as placeholders for arbitrary constants of our language, much as " ABC " above serves as a placeholder for arbitrary triangles. Thus, metavariables are not themselves part of our first-order language \mathcal{L} , but rather part of the extended English we are using to talk about the constants that *are* in the language. We will use the lower case *sans serif* characters a, b, c for this purpose.

Next are the variables, whose purpose will be clarified in detail below. The lower case letters x, y , and z , possibly with subscripts, will play this role, and we will suppose there to be an unlimited store of them. We will use the characters x, y and z as metavariables over the store of variables in our language.

Third, we have function symbols. These symbols correspond most closely in natural language to expressions of the form "The X of," where X is a common noun phrase like "color," "yearly salary," "mother," etc., or expressions of the form "The Y -est X in," where Y is an adjective like "smart" or "mean," and X once again by a common noun phrase. Common noun phrases typically express general properties. For any common noun phrase CNP , the result of replacing X with CNP in either of the above forms (together with an adjective for Y in the second form) intuitively names a function f that, when applied to a given object a , yields the appropriate instance $f(a)$ of the property expressed by the CNP for that object. Thus, where X is "color," the resulting function in the first form yields the color of the object to which it is applied; where it is "yearly salary," the resulting function yields an ap-

³The restriction to a finite number of constants here is not at all essential, but constraint languages in general will use only finitely many; the same holds for predicates and function names below.

appropriate dollar amount. Similarly, "The smartest woman in" expresses a function that takes places—e.g., cities, universities, etc.—and yields for each such place the smartest woman therein.

For the most part we will confine our attention to "one-place" functions such as those above that take a single object to another object. But as we will see there are occasions when we will want to represent functions of more than one argument as well. Examples of expressions that stand for two-place functions are "The only child of ... and ..." and "The sum of ... and" Intuitively, the former expresses a partial function⁴ from couples with a single child to that child, and the latter simply expresses the addition function, which takes two given numbers to a further number, viz., their sum.

As with constants, in practice it is often convenient to abbreviate relevant ordinary language functional expressions in defining the function symbols of a formal language. Again, we will use the letters f , g , and h , possibly with subscripts, for our basic function symbols, and corresponding *sans serif* characters as metavariables. Function symbols designed to stand for functions of more than one argument will be indicated with an appropriate numerical superscript. As above, we will suppose there are only finitely many of these symbols in our language.

We also introduce the symbol \bullet , and stipulate that where g stands for any n -place function symbol in our language, and f stands for any one-place function symbol, then $f \bullet g$ is an n -place function symbol as well. This corresponds in ordinary language to the fact that we can nest functional expressions, e.g., "The salary of the father of the smartest woman in largest university in ...," or "The successor of the sum of ... and"

The fourth group of symbols in our language consists of n -place predicates, $n > 1$. One-place predicates correspond roughly to verb phrases like "is a computer scientist," "has insomnia," "is an employee," and so forth, all of which express properties. Two-place predicates correspond roughly to transitive verbs like "loves," "is an element of," "is less than," "begat," and

⁴I.e., a function that might not be defined on every element of its domain. E.g., the square root function is only a partial function on the natural numbers, since it is not defined on those numbers which are not squares of other numbers. The function in the text here is partial because its intuitive domain is the set of pairs of humans, and not every such pair has a single child.

"lives with," which express two-place *relations* between things. There are also three-place relations, such as those expressed by "gives" and "between," and with a little work we could come up with relations of more than three places, but in practice we shall have little cause to go much beyond this.

We will use upper case roman letters such as P , Q , and R for predicates, and again corresponding *sans serif* characters as metavariables over predicates. Occasionally predicates will appear with numerical superscripts to indicate the number of places of the relation they represent, and if necessary with subscripts to distinguish those with the same superscripts. It is often useful to abbreviate relevant natural language expressions. Most languages contain a distinguished predicate for the two-place relation "is identical to." We will use the symbol \approx for this purpose.

To drive home the difference at this point between predicates and function symbols, note that a function symbol combines with names to yield yet another name-like (i.e., referring) expression: e.g., to draw on ordinary language, the function symbol "the husband of" combines with the name "Di" to yield the new referring expression (or *definite description*, as such are often called) "the husband of Di." On the other hand, a (one-place) predicate combines with a name to form a *sentence*, something that can be true or false, not a name-like expression. Thus, the predicate expression "is happy" combines with the name "Di" to yield the sentence "Di is happy." The same is easily seen to hold for n -place predicates generally.

The last group of symbols consists of the basic logical symbols: \neg , \wedge , \vee , \supset , \equiv , the existential quantifier \exists , and the universal quantifier \forall , about which we shall have more to say shortly. We will also need parentheses and perhaps other grouping indicators to prevent ambiguity.

2.2 Grammar

Now that we have our basic symbols, we need to know how to combine them into grammatical units, or *well-formed formulas*, the formal correlates of sentences. These will be the expressions that can encode the sort of information we will want to express in our theory (and more). This is done *recursively*

as follows.⁵

First, we want to group all name-like objects into a single category known as *terms*. This group will of course include the constants, and for reasons below, it will include the variables as well. But recall the discussion of function symbols above. There we saw that an expression like "The yearly salary of" seems to name a function on objects. But the values of functions are objects as well. Thus, when we attach a name, "Fred," say, to the functional expression above, the result—"The yearly salary of Fred"—is a sort of name for Fred's yearly salary. Thus, we count the result of attaching a functional symbol to an appropriate number of constants and/or variables as a term as well; and such terms can also be among the terms that a function symbol attaches to. Thus, more exactly, letting t_1, t_2, \dots stand for arbitrary terms and f stand for an arbitrary function symbol, if t_1, \dots, t_n are terms and f is an n -place function symbol, then $f(t_1, \dots, t_n)$ is a term as well.

Terms formed out of certain familiar two-place function symbols, examples of which will be introduced below, are more commonly written in *infix* notation, rather than the prefix notation just defined, with the function symbol flanked by the two terms, rather than preceding them. Thus, for a two-place function symbol f and terms t, t' , the term $f(t, t')$ can also be written as tft' . So, for example, $+(2, 3)$ can be written as $2 + 3$.

Next we define the basic formulas of our language. Just as verb phrases and transitive verbs in ordinary language combine with names to form sentences, so in our formal language predicates combine with terms to form formulas. Specifically, if P is any n -place predicate, and t_1, \dots, t_n are any n terms, then $Pt_1 \dots t_n$ is a formula, and in particular an *atomic* formula. To illustrate this, if H abbreviates the verb phrase "is happy," and a the name "Annie," then the formula Ha expresses the proposition that Annie is happy. Again, if L abbreviates the verb "loves," b the name "Bob," c the name "Charlie," and f the expression "the fiance of," then the formula $Lbf(c)$ expresses the proposition that Bob loves Charlie's fiance.

Often when one is using more elaborate predicates drawn from natural

⁵That is, the definition is given in such a way that complex cases of the class being defined are defined in terms of simpler cases of the same class. Recursive definitions thus often look circular, but they are not, as they always begin with well-grounded initial cases not defined in terms of other members of the class being defined.

language, e.g., if we had used *LOVES* instead of *L* in the previous example, it is more readable to use parentheses around the terms in atomic formulas that use the predicate and separate them by commas, e.g., *LOVES*(*b*, *x*) instead of *LOVESbx*. Thus, more generally, any atomic formula $Pt_1 \dots t_n$ can be written also as $P(t_1, \dots, t_n)$. Furthermore, atomic formulas involving some familiar two-place predicates like \approx , and a few others that will be introduced below, are more often written using infix rather than prefix notation. For example, we usually express that *a* is identical to *b* by writing $a \approx b$ rather than $\approx ab$. Thus, we stipulate that formulas of the form Ptt' can also be written as tPt' .

Now we begin introducing the logical symbols that allow us to build up more complex formulas. Intuitively, the symbol \neg expresses negation; i.e., it stands for the phrase "it is not the case that." Since we can negate any declarative sentence by attaching this phrase to the front of it, we have the corresponding rule in our formal grammar that if φ is any formula, then so is $\neg\varphi$. The symbols \wedge , \vee , \supset , and \equiv stand roughly for "and," "or," "if...then," and "if and only if," which are also (among other things) operators that form new sentences out of old in the obvious ways. Unlike negation, though, each takes *two* sentences and forms a new sentence from them. Thus, we have the corresponding rule that if φ and ψ are any two formulas of our language, then so are $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \supset \psi)$, and $(\varphi \equiv \psi)$.

Finally, we turn to the quantifiers \exists and \forall . Recall that we introduced variables without explanation above. Intuitively, \exists and \forall stand for "some" and "every," respectively; the job of the variables is to enable them to play this role in our formal language. Consider the difference between "Annie is happy," "Some individual is happy," and "Every individual is happy." In the first case, a specific individual is picked out by the name "Annie" and the property of being happy is predicated of her. In the second, all that is stated is that some unspecified individual or other has this property. And in the third, it is stated that every individual, whether specifiable or not, has this property. This lack of specificity in the latter two cases can be made explicit by rephrasing them like this: for some (resp., every) individual *x*, *x* is happy. Since the rule for building atomic formulas counted variables among the terms, we have the means for representing these paraphrases. Let *H* abbreviate "is happy" once again; then we can represent the paraphrases

as $\exists x Hx$ and $\forall x Hx$ respectively.

Accordingly, we add the final rule to our grammar: if φ is any formula of our language and x is any variable, then $\exists x\varphi$ and $\forall x\varphi$ are formulas as well. In such a case we say that the variable x is *bound* by the quantifier \exists (resp., \forall), and we say that the formula φ is the *scope* of the quantifier \exists in $\exists x\varphi$, and it is the scope of the quantifier \forall in $\forall x\varphi$.

3 First-order Semantics

3.1 Structures and Interpretations

We have motivated the construction of our grammar by referring to the intended meanings of the logical symbols and by letting our constants and variables abbreviate meaningful expressions out of ordinary language. But from a purely formal point of view, all we have in a language is uninterpreted syntax; we have not described in any formal way how to assign meaning to the elements of a first-order language. We will do so now.

A *structure* for a first-order language \mathcal{L} consists simply of two elements: a set \mathcal{D} called the *domain* of the structure, and a function \mathcal{F} known as an *interpretation function* for \mathcal{L} . Intuitively, \mathcal{D} is the set of things one is describing with the resources of \mathcal{L} , e.g., the natural numbers, major league baseball teams, the people and objects that make up an air force base, or the records inside a database. The purpose of \mathcal{F} is to fix the meanings of the basic elements of \mathcal{L} in terms of objects in or constructed from \mathcal{D} .

3.1.1 Interpretations of Constants and Function Symbols

The interpretation function works like this. First we deal with terms. We begin by noting that variables will not receive an interpretation, since their meanings can vary (they are *variables* after all) within a structure. They will be treated with their own special semantic apparatus below. Constants on the other hand, being the formal analogues of names with fixed meanings, are assigned members of \mathcal{D} once and for all as their interpretation; in symbols, for all constants κ of \mathcal{L} , $\mathcal{F}(\kappa) \in \mathcal{D}$.

To deal with terms formed from function symbols, we need first to interpret the function symbols themselves. To begin with, each basic function symbol α is assigned a function $\mathcal{F}(\alpha)$ from \mathcal{D} into \mathcal{D} . As indicated above, the functions expressed in ordinary language are often *partial*; that is, they are often not defined everywhere. For example, the function expressed by "The salary of" is not defined when applied to a conveyer belt or a garden vegetable. This suggests that we ought to let the functions from \mathcal{D} into \mathcal{D} that interpret our function symbols be partial. This leads to certain inelegancies in our formal apparatus, however, so we opt instead to include a distinguished object \perp in our domain \mathcal{D} whose sole purpose is to be the value of functions applied to objects on which they are intuitively undefined. Thus, if we have a function symbol f abbreviating "The salary of," and if our domain \mathcal{D} contains both persons and conveyer belts, then the interpretation of f will be the function that takes each person to his or her salary in dollars, and every other kind of object to our distinguished object \perp . Formally, then, for all basic n -place function symbols α of \mathcal{L} , $\mathcal{F}(\alpha) \in \{\Phi \mid \Phi : \mathcal{D}^n \longrightarrow \mathcal{D}\}$; that is, the interpretation of a basic n -place function symbol α of \mathcal{L} is going to be an element of the set of all n -place functions from the set of n -tuples of the domain \mathcal{D} into \mathcal{D} .

Now we need to address the nonbasic function symbols, i.e., those of the form $\alpha \bullet \beta$ which correspond to nested functional expressions in ordinary language like "The salary of the father of." Intuitively, we want $\mathcal{F}(\alpha \bullet \beta)$ to be the composition of $\mathcal{F}(\beta)$ with $\mathcal{F}(\alpha)$, i.e., $\mathcal{F}(\alpha) \circ \mathcal{F}(\beta)$, where in general $(\Phi \circ \Psi)(x) = \Phi(\Psi(x))$ ⁶—in terms of our example, the composition of the function expressed by "The salary of" with the function expressed by "the father of." Notice that by our trick with \perp , the composition of any two functions will always be total.

3.1.2 Interpretations of Predicates

Finally, for any one-place predicate P , we let $\mathcal{F}(P)$ be a subset of \mathcal{D} —intuitively, the set of things that have the property expressed by P . And for any n -place predicate R , $n > 1$, we let $\mathcal{F}(R)$ be a set of n -tuples of ele-

⁶Note that \circ is a *metalinguistic* symbol of our extended English that expresses the meaning of our object language symbol \bullet , viz., the composition function.

ments of \mathcal{D} —intuitively, the set of n -tuples of objects in \mathcal{D} that stand in the relation expressed by R . Thus, for example, if we want L to abbreviate the verb “loves,” then if our domain \mathcal{D} consists of the population of Texas, then $\mathcal{F}(L)$ will be the set of all pairs $\langle a, b \rangle$ such that a loves b . Formally, then, for all n -place predicates P , $\mathcal{F}(P) \subseteq \mathcal{D}^n$.⁷

If one wishes to include the identity predicate \approx in one’s language, and have it carry its intended meaning, then one needs an additional, more specific semantical rule designed to do this. Identity, of course, is a relation that holds between any object and itself, but not between itself and any other object. This additional semantical constraint is easy to express formally: if our language \mathcal{L} contains \approx , then the interpretation of \approx is the set of all pairs $\langle o, o \rangle$ such that o is an element of the domain \mathcal{D} , i.e., more formally, $\mathcal{F}(\approx) = \{\langle o, o \rangle \mid o \in \mathcal{D}\}$.

3.2 Truth

3.2.1 Variable Assignments

Given a structure $M = \langle \mathcal{D}, \mathcal{F} \rangle$ for \mathcal{L} (cf. the definition at the beginning of Section 3.1) we can define what it is for a formula of \mathcal{L} to be *true* in M . As usual, this is done recursively. First we need to introduce the notion of an *assignment* α for the variables, which is a sort of addendum to our interpretation function: it assigns members of the domain to variables. Relative to an assignment function α , we can define the interpretation of a complex term $f(t_1, \dots, t_n)$, for any function symbol f and any terms t_1, \dots, t_n . An interpretation function \mathcal{F} alone does not suffice for this since complex functional terms might contain variables, e.g., the term $f(x)$, which are ignored by interpretation functions. But if we supplement \mathcal{F} with an assignment α for the variables, then we have something for the function $\mathcal{F}(f)$ to work on. Specifically, the interpretation of the term $f(x)$ *under* α , $\mathcal{F}_\alpha(f(x))$, is just the function $\mathcal{F}(f)$ applied to $\alpha(x)$, the value assigned to x by α .

⁷Where $\mathcal{D}^1 = \mathcal{D}$, and $\mathcal{D}^{n+1} = \mathcal{D}^n \times \mathcal{D}$; i.e., \mathcal{D}^1 is just \mathcal{D} itself, \mathcal{D}^2 is the set of all pairs of members (i.e., the Cartesian product $\mathcal{D} \times \mathcal{D}$) of \mathcal{D} , \mathcal{D}^3 the set of all triples of members of \mathcal{D} , and in general \mathcal{D}^n is the set of all n -tuples of members of \mathcal{D} .

In general, then, let \mathcal{F}_α be the result of adding α to \mathcal{F} .⁸ Then the interpretation $\mathcal{F}_\alpha(f(t_1, \dots, t_n))$ of a complex term $f(t_1, \dots, t_n)$ under α is simply the result of applying the function $\mathcal{F}_\alpha(f)$ (which is just $\mathcal{F}(f)$, since f is a function symbol) to the objects $\mathcal{F}_\alpha(t_1), \dots, \mathcal{F}_\alpha(t_n)$, i.e., $\mathcal{F}_\alpha(f)(\mathcal{F}_\alpha(t_1), \dots, \mathcal{F}_\alpha(t_n))$.

3.2.2 Truth Under an Assignment

Atomic Formulas Our goal in this section is to define the notion of a formula being *true* in a structure M . To do so, we will first define a closely related notion, viz., that of truth *under an assignment* α . For convenience, we will sometimes speak of a formula being “true _{α} in M ” instead of being “true in M under α .” We start by defining this notion for atomic formulas. So let φ be an atomic formula $Pt_1 \dots t_n$. Then φ is true _{α} in M just in case $\langle \mathcal{F}_\alpha(t_1), \dots, \mathcal{F}_\alpha(t_n) \rangle \in \mathcal{F}_\alpha(P)$. Intuitively, then, where $n = 1$, Pt is true _{α} in M just in case the object in \mathcal{D} that t denotes is in the set of things that have the property expressed by P . And for $n > 1$, $Pt_1 \dots t_n$ is true _{α} just in case the n -tuple of objects $\langle o_1, \dots, o_n \rangle$ denoted by t_1, \dots, t_n respectively is in the set of n -tuples whose members stand in the relation expressed by P , i.e., just in case those objects stand in that relation.

Let us actually construct a small language \mathcal{L}^* and build a small structure M^* to illustrate these ideas. Suppose we have four names a, b, c, d , a single function symbol h (intuitively, to abbreviate “the husband of”), a one-place predicate H (intuitively, to abbreviate “is happy”), and a three-place predicate T (intuitively, to abbreviate “is talking to ... about”). Let us also include the distinguished predicate \approx , though we will make no real use of it until later. We will use x, y , and z for our variables.

For our structure M^* , we will take our domain \mathcal{D} to be a set of three individuals, $\{\text{Beth}, \text{Charlie}, \text{Di}\}$, and our interpretation function \mathcal{G} will be defined as follows. For our constants, $\mathcal{G}(a) = \mathcal{G}(b) = \text{Beth}$, $\mathcal{G}(c) = \text{Charlie}$, and $\mathcal{G}(d) = \text{Di}$. (Beth thus has two names in our language; this is to illustrate a point to be made several sections hence.) For our function symbol h , we let $\mathcal{G}(h)(\text{Beth}) = \mathcal{G}(h)(\text{Charlie}) = \perp$ (so that $\mathcal{G}(h)$ is “undefined” on Beth and Charlie), and $\mathcal{G}(h)(\text{Di}) = \text{Charlie}$. For our predicates H and T ,

⁸I.e., if ξ is a constant, function symbol, or predicate, $\mathcal{F}_\alpha(\xi) = \mathcal{F}(\xi)$, and if ξ is a variable, then $\mathcal{F}_\alpha(\xi) = \alpha(\xi)$.

we let $\mathcal{G}(H) = \{\text{Beth}, \text{Di}\}$ (so, intuitively, Beth and Di are happy), and $\mathcal{G}(T) = \{(\text{Beth}, \text{Di}, \text{Charlie}), (\text{Charlie}, \text{Charlie}, \text{Di})\}$ (so, intuitively, Beth is talking to Di about Charlie, and Charlie is talking to himself about Di). Following the rule for \approx , we let $\mathcal{G}(\approx) = \{(\text{Beth}, \text{Beth}), (\text{Charlie}, \text{Charlie}), (\text{Di}, \text{Di})\}$. Finally, for our assignment function β , let us let $\beta(x) = \beta(y) = \text{Charlie}$, and $\beta(z) = \text{Di}$.

Let us now check that Hd and $Tbdh(z)$ are true in M^* under β . In the first case, by the above, Hd is true_β in M^* just in case $\mathcal{G}_\beta(d) \in \mathcal{G}_\beta(H)$, i.e., just in case Di is an element of the set $\{\text{Beth}, \text{Di}\}$, which she is. So Hd is true_β in M^* . Similarly, $Tbdh(z)$ is true_β in M^* just in case $\langle \mathcal{G}_\beta(b), \mathcal{G}_\beta(d), \mathcal{G}_\beta(h(z)) \rangle \in \mathcal{G}_\beta(T)$, i.e., just in case $\langle \mathcal{G}(b), \mathcal{G}(d), \mathcal{G}(h)(\beta(z)) \rangle \in \mathcal{G}(T)$, i.e., just in case $\langle \text{Beth}, \text{Di}, \mathcal{G}(h)(\text{Di}) \rangle \in \{(\text{Beth}, \text{Di}, \text{Charlie}), (\text{Charlie}, \text{Charlie}, \text{Di})\}$ i.e., just in case $\langle \text{Beth}, \text{Di}, \text{Charlie} \rangle \in \{(\text{Beth}, \text{Di}, \text{Charlie}), (\text{Charlie}, \text{Charlie}, \text{Di})\}$. Since this obviously holds, the formula $Tbdh(z)$ is true_β in M^* .

A formula is false_α in a structure M , of course, just in case it is not true_α in M . It is easy to verify that, for example, $Hh(b)$, Hx , and $Tdbc$ are all false_β in M^* under β .

Conjunctions, Negations, etc. Now for the more complex cases. Suppose first that φ is a formula of the form $\neg\psi$. Then φ is true_α in a structure M just in case ψ is *not* true_α in M . In so defining truth for negated formulas we ensure that the symbol \neg means what we have intended. Things are much the same for the other symbols. Thus, suppose φ is a formula of the form $\psi \wedge \theta$. Then φ is true_α in M just in case both ψ and θ are. If φ is a formula of the form $\psi \vee \theta$, then φ is true_α in M just in case either ψ or θ is. If φ is a formula of the form $\psi \supset \theta$, then φ is true_α in M just in case either ψ is false in M or θ is true_α in M . And if φ is a formula of the form $\psi \equiv \theta$, then φ is true_α in M just in case ψ and θ have the same truth value in M .

The reader should test his or her comprehension of these rules by verifying that $\neg Hh(b)$ and $(Tbdh(z) \wedge Tccy) \supset Hd$ are both true in M^* under β .

Quantified Formulas Last, we turn to quantified formulas. When we introduced the quantifiers above, we noted that "Some individual is happy," i.e., $\exists x Hx$, can be paraphrased as "for some value of the variable 'x,' the expression 'x is happy' is true." This is essentially what our formal seman-

tics for existentially quantified formulas will come to. To anticipate things a bit, $\exists x Hx$ will be true in a structure M under α , roughly, just in case the unquantified formula Hx is true in M under some (in general, new) assignment α' such that $\alpha'(x)$ is in the interpretation of H . It is easy to verify that this formula is true in our little structure M^* under β , when we look at a new assignment function β' that assigns either Beth or Di to the variable x . Thus, $\exists x Hx$ should come out true in M^* under β .

But we have to be a little more careful, because some formulas— $Tcxz$, for example—contain more than one unquantified variable. Thus, when we are evaluating a quantification of such a formula— $\exists z Tcxz$, say—we have to be sure that the new assignment function α' does not change the value of any of the unquantified variables—in this case, the variable x . Otherwise we could change the sense of the unquantified formula in mid-evaluation. Under the assignment function β above, $\exists z Tcxz$ intuitively says that Charlie is talking to himself about someone (recall that $\beta(x) = \text{Charlie}$), and this should turn out to be true $_{\beta}$ in M^* since Charlie is talking to himself about Di, i.e., $\langle \text{Charlie}, \text{Charlie}, \text{Di} \rangle \in \mathcal{G}_{\beta}(T)$. But suppose all we require is that there be some new assignment function β' such that $\beta'(z)$ is Di. Then it could turn out also that $\beta'(x)$ is Beth. But then the formula $Tcxz$ would not be true in M^* under β , since Charlie is not talking to Beth about Di, i.e., $\langle \text{Charlie}, \text{Beth}, \text{Di} \rangle \notin \mathcal{G}_{\beta}(T)$, and hence we would not be able to count $\exists z Tcxz$ as true in M^* under β after all as we should like.

All that is needed is a simple and obvious restriction: when evaluating the formula $\exists z Tcxz$, the new assignment function that we use to evaluate $Tcxz$ must not be allowed to differ from β on any variable except z (and even then it *needn't* differ from β ; in which case it *is* β). More generally, we put the matter like this: if φ is an existentially quantified formula $\exists x \psi$, then φ is true in a structure M under α just in case there is an assignment function α' just like α except perhaps in what it assigns to x such that the formula ψ is true in M under α' . If φ is a universally quantified formula $\forall x \psi$, then φ is true in M under α just in case for *every* assignment function α' just like α except perhaps in what it assigns to x the formula ψ is true in M under α' . That is, in essence, φ is true in M just in case ψ is true in M no matter what value in the domain we assign to x (while keeping all other variable assignments fixed).

The reader can once again test his or her comprehension by showing in detail that $\exists x T x b h(z)$ is false in M^* under β and that $\forall x (H x \vee T b d x)$ is true in M^* under β .

3.2.3 Truth

Now, finally, we can define a formula to be *true* in a structure M *simpliciter* just in case it is true_α in M for all assignments α , and *false* in M just in case it is false_α in M for all α . Note, on this definition, that for most any interpretation, there will be formulas that are neither true nor false in the interpretation. Our example $\exists z T b x z$ above, for instance, is neither true nor false in M^* , since there are assignments α on which it comes out true_α —all those on which $\alpha(x) = \text{Di}$ —and assignments α on which it comes out false_α —all those on which $\alpha(x) \neq \text{Di}$. Such formulas will always have free variables, since it is the semantic indeterminacy of such variables that is responsible for this fact. However, note that some formulas with free variables will be true or false in some models, though these will typically be logical truths (or falsehoods) like $H x \wedge \neg H x$, i.e., formulas which are not capable of true (resp., false) interpretation.

4 Logic

4.1 Propositional Logic

Now that we have the notion of a first-order language and its semantics, we want to capture the meanings of the logical constants \neg , \wedge , \vee , \supset , \equiv , \forall , and \exists as explicated in the semantics. We will do this in the usual way by developing a rigorous and precise *logic*. A logic, in the sense relevant here, is a systematic characterization of correct principles of reasoning with respect to a given cluster of concepts. The concepts here are those expressed by the logical constants above, corresponding roughly, once again, to the ordinary language concepts of negation (*not*, or *it is not the case that*), conjunction (*and*), disjunction (*or*), material implication (*if ... then*), material equivalence (*if and only if*), existential quantification (*some*), and universal quantification (*every*, or *all*). The form such a system takes usually consists

of two components: *axioms* and *rules of inference*. We start with the axioms for the propositional connectives.

4.1.1 Axioms for Propositional Connectives

The axioms for the propositional connectives \neg , \wedge , \vee , \supset , and \equiv constitute the basis of *propositional logic* and can be thought of as characterizing their meanings. There are many equivalent axiomatizations for propositional logic, but the following, which makes use of the notion of an axiom *schema*, is one of the easiest. An axiom schema is not itself an axiom, but rather a sort of template, a general form any *instance* of which is an axiom. Axiom schemas are thus not themselves actually part of the language. Thus, where φ , ψ , and θ are any formulas, any instance of any of the following schemas is an axiom:

$$\mathbf{A1} \quad \varphi \supset (\psi \supset \varphi)$$

$$\mathbf{A2} \quad (\varphi \supset (\psi \supset \theta)) \supset ((\varphi \supset \psi) \supset (\varphi \supset \theta))$$

$$\mathbf{A3} \quad (\varphi \supset \psi) \supset ((\neg \varphi \supset \psi) \supset \psi)$$

In English, **A1** says essentially that if a sentence φ is true, then for any other sentence ψ , if ψ is true then φ is still true. **A2** says that if a sentence φ implies that if ψ is true then so is θ , then if φ implies ψ , then it also implies θ . Finally, **A3** says essentially that if a sentence φ implies another sentence ψ , then if ψ is also implied by the negation of φ , then ψ is true no matter what (since either φ or its negation is true no matter what). These axioms seem trivial. However, like the elementary truths of arithmetic or geometry that are second nature to us now, they must be explicitly stated as a basis for deriving other, less obvious truths; they cannot be conjured out of thin air.

Notice that axiom schemas only use the two connectives \neg and \supset . Even though we have been using the other propositional connectives all along, officially we will consider these to be our two “primitive” connectives; the others can be defined in terms of them as follows (where the symbol $=_d$ means “is defined as”):



Def 1: $(\varphi \vee \psi) =_{df} (\neg\varphi \supset \psi)$

Def 2: $(\varphi \wedge \psi) =_{df} \neg(\neg\varphi \vee \neg\psi)$

Def 3: $(\varphi \equiv \psi) =_{df} (\varphi \supset \psi) \wedge (\psi \supset \varphi)$

The reader can again test comprehension by showing that, no matter what truth values are assigned to φ , ψ , and θ , the two sides of each definition will always have the same truth values when evaluated in accord with the semantical rules given above for the connectives in Section 3.2.2.

4.1.2 Rules of Inference: Modus Ponens

A logic is not much good without rules of inference, which are rules that allow us to move from statements that we know or assume to be true at the outset (e.g., our axioms), to new statements that follow logically from them (called *theorems*). Without them, all we could do is write down axioms; there would be no way to infer new truths from those already given. There is only one rule of inference in propositional logic:

Modus Ponens (MP): If the formulas φ and $\varphi \supset \psi$ follow from the axioms of propositional logic, then we may infer that ψ does as well.⁹

As a simple example using our language \mathcal{L}^* , consider the following proof of $Hd \supset Hd$, i.e., the statement *If Di is happy, then Di is happy*. Note that, trivial as it is, $Hd \supset Hd$ is not an instance of an axiom schema, and hence if it is to be a theorem of our system, it must be derivable from the axioms using our rule of inference MP. This is in fact the case. As an instance of A1, we have

$$Hd \supset ((Hd \supset Hd) \supset Hd).$$

As an instance of A2 we have

$$(Hd \supset ((Hd \supset Hd) \supset Hd)) \supset ((Hd \supset (Hd \supset Hd)) \supset (Hd \supset Hd)).$$

⁹Given this, the notion of theoremhood can be defined precisely as follows. A formula φ is a theorem of propositional logic if and only if there is a sequence $\varphi_1, \dots, \varphi_n$ such that φ_n is φ and each φ_i is either an axiom or follows from previous lines by MP, that is, there are previous formulas φ_j, φ_k , $j, k < i$, such that φ_i is $\varphi_j \supset \varphi_k$. We can also define the notion of a formula ϕ following from a set of formulas Γ in the same way except by adding in addition that ϕ_i in the above definition could also be a member of Γ .

By MP, it follows from these two statements that

$$(Hd \supset (Hd \supset Hd)) \supset (Hd \supset Hd).$$

But

$$(Hd \supset (Hd \supset Hd))$$

is an instance of A1 again, hence by MP once more we can infer $Hd \supset Hd$ from the latter two statements.

There are many equivalent systems of propositional logic that are more streamlined and computationally more efficient than the basic system here; but this is the foundation on which they are all built and illustrates well enough how the process of deduction works.

4.2 Predicate Logic

4.2.1 Axioms for the Quantifiers

When we add axioms for the quantifiers to propositional logic, we have full *predicate logic*, also known as *first-order logic* and *quantification theory*. The quantifiers are interdefinable, so we only need to take one of them as primitive. The axioms for predicate logic are usually stated in terms of the universal quantifier \forall , so we will take that as our primitive, and shall define \exists as follows:

Def 4: $\exists x\varphi =_{df} \neg\forall x\neg\varphi$.

That this definition is correct is clear on a moment's reflection. So, for example, there exists an x such that x is happy, i.e., someone is happy, just in case it is not that case that for all x , x is not happy, i.e., just in case not everyone is unhappy.

We can now state three new quantificational axiom schemas. For any formula φ and term t , we let φ_t^x stand for the result of substituting all unbound occurrences of x in φ with t . Then any instance of the following is an axiom:

A4 $\forall x\varphi \supset \varphi_t^x$, so long as t does not contain, and is not itself, a variable that becomes bound in φ_t^x .

A5 $\forall x(\varphi \supset \psi) \supset (\forall x\varphi \supset \forall x\psi)$.

A6 $\varphi \supset \forall x\varphi$, where x does not occur unbound in φ .

The intuitive idea behind these axioms is straightforward. **A4** simply says that if something is true of everything in general, it is true in particular of anything we can name. Thus, for example, $\forall x(NUM(x) \supset \exists y(y = x + 1)) \supset (NUM(24) \supset \exists y(y = 24 + 1))$; i.e., if for every number there is a number one greater than it, then in particular there is a number one greater than 24.

Reverting to our language \mathcal{L}^* and its structure M^* , we have as an instance of this axiom schema

$$\forall x(Hx \vee \exists yTxy) \supset (Hc \vee \exists yTccy).$$

The antecedent here (i.e., the formula to the left of the \supset), $\forall x(Hx \vee \exists yTxy)$, is in fact true in M^* , i.e., in M^* , everyone is either happy or talking to themselves about someone in M^* . Thus, if we were to count this as a further “special” axiom—i.e., a nonlogical piece of information that characterizes the situation in the specific structure we are investigating and which might well not hold in other structures—we would be able to prove (by Modus Ponens) that $(Hc \vee \exists yTccy)$, i.e., that Charlie is either happy or talking to himself about someone.

The second schema **A5** captures another aspect of the meaning of “every.” Consider a simple example: if every individual is such that if it is red then it has a color, then if in fact every individual is red, then every individual has a color. This is just an unsymbolized instance of **A5**, and illustrates its validity.

And finally, **A6** simply says that a quantifier does not affect the truth of a formula φ if the quantifier does not bind a variable that does not occur in φ or—what amounts to the same thing—occurs in φ but is bound by another quantifier. So, for example, if it is true that Beth is happy, Hb , then it is also true for every value of x that Beth is happy, $\forall xHb$. Similarly, if Charlie is talking to someone about Di, $\exists zTczd$, then it is also true that for every value of x , Charlie is talking to someone about Di, $\forall x\exists zTczd$.

4.2.2 Rules of Inference: Generalization

The move to predicate logic with its quantified formulas necessitates a further rule of inference, one designed to capture how we reason with universal quantification. As usual, the idea is best illustrated by an example. Suppose you wanted to prove something about all prime numbers, for example, that for every prime there is a greater prime. You might begin by saying something like "Let p be an arbitrary prime number." You might even pick a specific prime, for example, 17. Then, by appealing to none of the specific properties of your chosen prime that distinguish it from other primes, e.g., that it is less than 100, or Plato's favorite number, etc., you proceed to prove in the usual way that there is another prime greater than p . You then conclude that the same is true for *every* prime. What permits you to do this is precisely the fact that you did not appeal to any properties of p that do not hold for all primes; it was, in a precise sense, arbitrary.

This sort of example illustrates the inference rule known as *Generalization*. Informally, if you can prove that something is true of a particular individual o without appealing to anything that could not be proved of everything else in the domain, then that same thing is true of everything. The way we capture this idea of not appealing to anything that could not be proved of everything else is by restricting generalization to formulas whose proofs contain no formulas that say anything about the object being generalized upon. Thus, we can say that if ϕ^x follows from Γ and the axioms of predicate logic, and t does not occur (free) in Γ , then $\forall x\phi$ follows from Γ and the axioms of predicate logic. If, then, t refers to the object o , then the absence of t from the formulas in Γ indicates that they say nothing about o . In fact, we can actually use a simpler but equivalent inference rule that only generalizes on variables:

Gen If φ follows from the axioms of predicate logic, then $\forall x\varphi$ does as well.

We noted above that special, or nonlogical, axioms are designed only to hold within a given structure one has singled out, e.g., a structure that models a certain manufacturing or engineering system one might be investigating. A special axiom thus captures the "logic" things within a restricted sphere. Genuine logical axioms, however, should be exceptionless; a logical axiom

formulated within a given language \mathcal{L} should be true in all structures of \mathcal{L} . When this property holds of all the axioms of a logical system, the system is said to be *sound*. Soundness is an essential property of any logical system, since it is precisely the job of its logical axioms to capture features that hold in any of its structures. Any axiom that was not true in every structure could therefore not rightfully be considered a *logical* axiom, and would have to be rejected. It is straightforward (and a good exercise) to show that, for a given language \mathcal{L} , any instance of any of the above axiom schemas, and anything provable from them, in fact has this property.¹⁰

The converse of soundness, that any formula true in every structure follows from the axioms, is known as *completeness*, and is much harder to prove. While its absence from a formal system is perhaps not as disastrous as the absence of soundness, completeness is nonetheless a very important and desirable property for a formal system to have, since it shows that the semantics and the logic of the system match up precisely. It is provable that both propositional and predicate logic are complete.

4.3 Identity

4.3.1 Identity and Expressive Power

A very important concept within most any type of formal system is that of *identity*, which we will express in our languages by means of the 2-place predicate \approx .¹¹ Identity adds a great deal of flexibility and expressive power to a language. Identity is particularly useful in languages that contain function symbols, for with identity one can explicitly identify a named object as the value of a certain function. For example, in our language \mathcal{L}^* , we can express that Charlie is Di's husband, $c \approx h(d)$.

Second, identity can be used to express the definite article "the." When we ascribe a property to something only identified as "the φ "—that *the person Charlie is talking to himself about* is happy, say—we are implying

¹⁰The proof proceeds by ordinary mathematical induction on the number of quantifiers and connectives a formula contains.

¹¹We use \approx as our identity predicate *within* languages; this is to be distinguished from the concept of identity as it appears in our metalinguistic talk *about* languages and their structures, which we have been expressing with the more familiar $=$.

three things: (i) that there is something that fits the description φ —that there is someone Charlie is talking to himself about—(ii) that nothing else fits it—that Charlie is not talking to himself about anyone else—and (iii) that that thing has the property in question—that the object of Charlie's attention is happy.¹² All three of these components are easily expressed in one formula with the help of the identity predicate. Thus, our example here is expressed in our language \mathcal{L}^* as follows: $\exists x(Tccx \wedge \neg \exists y(Tccy \wedge x \neq y) \wedge Hx)$. The force of the "anyone else" in (ii) above here is captured by the negated identity predicate here in the formula: anyone *other than*, i.e., not *identical to*, the person in question.

Finally, similar techniques can be employed to express numerical notions without appealing explicitly to numbers. For example, one can express that *at least two* philosophers are wealthy as $\exists x \exists y (Px \wedge Py \wedge x \neq y)$. Note that the third conjunct here is necessary, since the bare statement $\exists x \exists y (Px \wedge Py)$ does not imply there are *two* wealthy philosophers—both x and y could be assigned the same unique wealthy philosopher as their values (convince yourself of this by referring back to the section on the semantics of \exists). In a similar fashion, one can express that there are *exactly two* wealthy philosophers: $\exists x \exists y (Px \wedge Py \wedge x \neq y \wedge \forall z (Pz \supset (z \approx x \vee z \approx y)))$, i.e., in English, there are at least two wealthy philosophers x and y , and any wealthy philosopher is identical with either x or y . Finally, one can also say that there are *at most two* wealthy philosophers: $\forall x \forall y \forall z ((Px \wedge Py \wedge Pz) \supset (x \approx y \vee x \approx z \vee y \approx z))$. Check to see that this statement will be true if there are fewer than three philosophers, and false otherwise. These forms are easily generalizable for any finite number.

4.3.2 Axioms for Identity

Most systems of predicate logic include the notion of identity among the logical constants of the system. Given one standard (though debatable) conception of logic as the study of the *most general* principles of reasoning, this seems quite appropriate, since identity is a notion that seems applicable to most any domain about which one might reason. Irrespective of the issue

¹²This is the essence of Bertrand Russell's *theory of descriptions*, first developed in his famous paper "On Denoting," *Mind* 14 (1905).

of whether identity is a logical notion, it is certainly a notion one might often want to use within a formal system that has been tailored for a certain purpose, and in particular, it is essential to our constraint languages. However, the only way to ensure that the identity predicate carries its intended meaning within a given system is to build that meaning into the system by means of appropriate axioms. The usual axioms for identity are, as above, presented in the form of schemas, and are also straightforward:

A7 $t \approx t$, for any term t

A8 $x \approx t \supset (\varphi \supset \varphi_t^x)$, so long as t does not contain, and is not itself, a variable that becomes bound in φ_t^x .

A7 captures the point made above, that identity holds between any object and itself. A8 is nearly as intuitive. The idea is simply that if something is true of a given object, then it does not matter how the object is referred to; it is still true of it.¹³ If, for example, Mark Twain wrote *Huckleberry Finn*, then it follows that Samuel Clemens did as well, since they are the same person. That is, more formally, by A8 it is an axiom that

$$m \approx s \supset (WROTE(m, h) \supset WROTE(s, h)).$$

If again we add $m \approx s$ as a special axiom, or derive it from other information we possess, we can then prove by MP that $WROTE(m, h) \supset WROTE(s, h)$. If we then have in addition the further information that $WROTE(m, h)$, we can prove by MP once again that $WROTE(s, h)$.

As a second example, let us revert to our language \mathcal{L}^* once again, in

¹³There are well known exceptions to this. For example, suppose Shorty is five feet tall, and that his real name is "Eddie." So Shorty \approx Eddie. Nonetheless, from the fact that Shorty is so-called because of his size, it does not follow that Eddie is so-called because of his size. Other famous contexts where this principle seems to break down are those involving psychological attitudes like belief. For example, even though I believe that 9 is prime, I may not, due to my rusty calculus, believe that $\int_0^3 x^2 dx$ is prime, despite the fact that $\int_0^3 x^2 dx \approx 9$. In the semantics and logic we are constructing it is assumed that we shall not be needing to formalize expressions like "is so-called because of" and "believes"—though it should be noted that the apparatus we have developed here is eminently capable of being extended to handle such expressions.

which we included the identity predicate. In that language, we have both

$$a \approx c \supset (Ha \supset Hc)$$

and

$$a \approx b \supset (Ha \supset Hb)$$

as instances of **A6**. In M^* , $a \approx c$ is false, since $\mathcal{G}(a) = \text{Beth}$, and $\mathcal{G}(c) = \text{Charlie}$. Thus, $a \approx c$ would not be considered among any special axioms we might have to characterize M^* . Hence, as we should hope, we would not be able to infer $Ha \supset Hc$, which is also false in M^* . However, $a \approx b$ is true in M^* (recall that we assigned both a and b to Beth as their interpretation), and hence could be a special axiom for the situation characterized by our structure. By **MP** we could then infer from the second of the two instances above that $Ha \supset Hb$, and from Ha (which might be a further special axiom perhaps) that Hb .

As one would hope, our logic remains sound and complete when we add the axioms for identity.

5 Constraint Languages

Now that we have a well-developed logical foundation, we will begin to add the particular elements that constitute a constraint language. In actuality, there will be infinitely many possible constraint languages, since each set of predicates specifies a different language. However, all of them will have certain elements in common, and it is these common elements we want to begin laying out now.

First, every constraint language will be a first-order language as described above. Second, we will assume that a constraint language will contain the basic resources of arithmetic—a distinguished predicate *NUM*, the numerals, the usual function symbols $+$, \cdot , and *exp*, and enough axiomatic power to prove basic arithmetical facts. The intended semantics for any constraint language will thus always contain the natural numbers, with these syntactic items receiving the obvious interpretations. Third, every constraint language will contain a certain amount of set theory. Throughout this discussion we have been employing set theory in a rough and ready fashion in our

description of the model theory for first-order languages. In a constraint language, we will want to be able to do this in a principled way.

The full theory of sets that one might find in a text book is very powerful and very complex. However, the structures for which we are designing our constraint languages are all relatively simple; indeed, they are all finite, though we shall not need to assume this. Furthermore, we will not need much more than the simplest set theoretic operations and constructions to express what we want to express. Hence, all we need is enough set theory to meet these limited needs. We will provide this, along with some motivation and explication of the relevant concepts, in the next section.

5.1 Basic Set Theory

5.1.1 Membership

A set, intuitively, is just a collection of things which themselves may or may not be sets. Usually we pick out a set with the help of some predicate, e.g., the set of all prime numbers, or American citizens, or track and field events in the 1988 Olympics. But this is just for our benefit; any collection of things, even if they cannot be picked out by a common property, indeed even if they cannot be picked out by us in any way at all (as is the case, e.g., with most infinite collections of natural numbers), still form a set. We will see shortly that we have to be a little more careful than this about the sets we claim to exist; but this at least gets our intuitions going about what sorts of things sets are.

The most basic relation a thing can bear to a set is that it can be a *member*, or *element*, of the set. Thus, the number 17 is a member of the set of all primes; George Bush is a member of the set of American citizens; and the now unofficial race in which Ben Johnson beat Carl Lewis is a member of the set of track and field events that took place in the 1988 Olympics. This special relation is nearly always represented by the symbol \in , and as with all the two place set-theoretic relations we will introduce, we will use infix rather than prefix notation. Thus, we will write $a \in b$ rather than $\in ab$.

Logically, sets are just individuals like any others, and so we will use constants to stand for them. And since not everything is a set, we will

introduce a special predicate *SET* to abbreviate "is a set." Since it will often be convenient to say something general only about sets, we will set aside the letters *r*, *s*, and *t* (again, perhaps with subscripts and primes) to serve as special *set variables* that take only sets as values (and as before corresponding *sans serif* characters to serve as metavariables). This way, we will be able to say things about all and only sets without having to use the predicate "*SET*" explicitly. For example, suppose we want to say that the object *a* is a member of some set. Without these special set variables we would have to express this as $\exists x(SET(x) \wedge a \in x)$. With them, however, we can simply write this as $\exists s(a \in s)$. Similarly, if we want to express that every set is a member of some other set, without the set variables we have to write $\forall x(SET(x) \supset \exists y(SET(y) \wedge x \in y))$, whereas with them we can simply write $\forall r \exists s(r \in s)$. In general, and more abstractly, if *s* is any set variable that does not occur in a formula φ , then $\forall x(SET(x) \supset \varphi)$ is equivalent to $\forall s \varphi_s^x$ and $\exists x(SET(x) \wedge \varphi)$ is equivalent to $\exists s \varphi_s^x$ (where, once again, φ_s^x is the result of replacing every unbound occurrence of *x* in φ with an occurrence of *s*).

It frequently happens that we want to say something φ about some or all the members of a given set *s*. In our current grammar, this would be expressed as $\exists x(x \in s \wedge \varphi)$ or $\forall x(x \in s \supset \varphi)$ respectively. For convenience we allow that these forms can be abbreviated as $(\exists x \in s)\varphi$ and $(\forall x \in s)\varphi$ respectively.

5.1.2 Basic Set Theoretic Axioms

Russell's Paradox Sets combine and interact in many interesting ways, but for deep and historically significant reasons, not every way in which one might think. For this reason we need to set down clear principles that tell us precisely when such combinations and interactions can occur, and furthermore exactly what sets exist within a given domain. That is, we need some set theoretic axioms.

In case the reader is not convinced of this need, consider the following famous paradox, known as *Russell's paradox*, after the famous philosopher/logician Bertrand Russell who discovered it. As noted above, we often pick out sets in ordinary contexts by means of some predicate or (more gen-

erally) description that holds of all and only the members of the set. Thus, for example, one might want to consider the set of all Texans over thirty-five who drink beer by means of the description "Texan over thirty-five who drinks beer," or more formally, the description $TEXAN(x) \wedge age_of(x) > 35 \wedge DRINKS_BEER(x)$. Let us use the notation $\{x \mid TEXAN(x) \wedge age_of(x) > 35 \wedge DRINKS_BEER(x)\}$ to name this set, and in general the notation $\{x \mid \varphi\}$ to name the set of things that satisfy the description φ . Now, intuitively, one would think that any such description φ with a single unbound variable picks out a corresponding set comprising the things that fit the description. For after all, a set is just a collection of things; so in particular the collection satisfying a certain description is a set. Russell found that, intuitions to the contrary, this is not always so. Consider the description "set that does not have itself as a member," i.e., $s \notin s$. (Remember that s is a set variable.) Intuitively, there are all sorts of sets that satisfy this description: the set of horses is not a horse and hence is not a member of itself, the set of solar planets is not a planet, and so on. By the intuitive principle above, there is a set of all sets that satisfies this description, i.e., there is the set $r = \{s \mid s \notin s\}$. But now ask yourself: is r a member of itself or not? If it is, then since r is the set of all sets that are not members of themselves, it follows that it is *not* a member of itself after all. If on the other hand it is not a member of itself, then it satisfies the condition for membership in r , i.e., it actually *is* a member of itself. Either way we contradict ourselves. So there cannot be such a set as r after all, despite what our intuitions tell us.

The Axioms The lesson here is that not just any collection of things we is a set. Hence the need for axioms that do not get us into the same sort of trouble. For our purposes, we need surprisingly few: four axioms and one axiom schema. The first axiom, *extensionality*, tells us when two apparent sets are in fact identical, viz., when they have exactly the same members:

$$ST1 \quad \forall r \forall s (\forall x (x \in r \equiv x \in s) \supset r \approx s),$$

i.e., for all sets r and s , if for any object x , x is a member of r if and only if it is a member of s , then r and s are the same set.

The second axiom, *pairing*, is that any two objects (within a given do-

main) form a set:

ST2 $\forall x \forall y \exists s (s \approx \{x, y\}),$

where " $\{x, y\}$ " is a name for the set that contains exactly the objects denoted by x and y . (By extensionality there can be only one such set.) Thus, to make this proper, we need to add to our vocabulary the left and right braces $\{, \}$, and to our grammar the rule that if t_1, \dots, t_n are any terms, then the expression $\{t_1, \dots, t_n\}$ is a term as well.¹⁴

The next axiom declares that the *union* of any set r exists, i.e., the set whose elements are exactly the members of the members of r :

ST3 $\forall r \exists s \forall y (y \in s \equiv \exists t (t \in r \wedge y \in t)),$

in English, for any set r there exists a set s such that for any object y , y is a member of s if and only if there is a set t such that t is a member of r and the object y is a member of t . For a given set r , we will let $\bigcup r$ stand for the union of r . (\bigcup is thus a distinguished two-place function symbol, denoting the (partial) function that takes any set to its union.) We will usually write $r \cup s$ for $\bigcup \{r, s\}$.

When one set a is a subset of another b (i.e., when all the members of a are members of b) we express this with a distinguished predicate \subseteq as $a \subseteq b$. The fourth axiom says that the set of all subsets of any given set exists:

ST4 $\forall r \exists s \forall x (x \in s \equiv x \subseteq r),$

that is, for any set r there is a set s such that for any object x , x is a member of s just in case x is a subset of r . If $a \subseteq b$ and $a \neq b$, we say that a is a *proper* subset of b , and we express this as $a \subset b$. For any given set a , the set of all its subsets is called the *power set* of a . The (partial) function that takes each set to its power set will be denoted by the distinguished function symbol *pow*, and thus the power set of a will be denoted by *pow*(a).

¹⁴Strictly speaking, we can think of ourselves as adding infinitely many new function symbols f_1, f_2, \dots to our language, where each f_n is an n -place function symbol, each of which can by convention be rewritten using the brace notation. The rewritten form of each f_n is thus evident by the fact that there are n terms between the braces, e.g., $\{a, b, c\}$ is the rewritten form of $f_3 abc$.

Finally we come to our one set theoretic axiom *schema*, so-called because it actually stands for infinitely many axioms of the same general form, one for each formula of our language. It is called the axiom schema of *separation*, or *subsets*. The idea is quite simple: given a certain set a and some description φ in our language, we can separate out the set of all the members of a that satisfy the description. Formally, for any formula φ ,

$$\text{ST5}_\varphi \quad \forall r \exists s \forall x \in r (x \in s \equiv \varphi(x)),$$

where $\varphi(x)$ is the result of replacing any unbound variable in φ with x .¹⁵

Russell's Paradox Revisited Given the separation axiom schema we are able to reintroduce in a restricted form the notation for sets used in the brief discussion of Russell's paradox above. The paradox arises when one assumes one can generate sets arbitrarily with any given formula. Separation allows one to use arbitrary formulas only to form sets from the members of *previously given* sets, and this eliminates the problem; in this light, in Russell's argument, for any given set a already proved to exist, one is allowed to assume only the existence of the set $\{s \mid s \in a \wedge s \notin s\}$, and this causes no problems at all. Thus, we can safely add the following grammatical rule:¹⁶ if φ is any formula, t any term, and x any variable, then $\{x \mid x \in t \wedge \varphi\}$ is a term as well. Similar to what we allowed with certain types of quantified formulas, such terms can also be written as $\{x \in t \mid \varphi\}$.

¹⁵ Assuming of course x does not become bound in the process; if it does, we can always replace it in the above schema with a new variable not occurring in φ .

¹⁶ Or more cautiously, it appears that we can do so safely for all we can tell. Due to Gödel's famous *second incompleteness theorem*, there is no way to *prove* that there are not other hitherto undiscovered paradoxes lurking in the theory of sets; that is, we cannot prove its consistency (at least, not without begging the question by proving it in a theory that is at least as dubious). The great success of the theory over the past eighty-five years, however, and the absence of any new paradoxes despite extensive use and scrutiny of the theory, has given logicians great confidence that it is in fact consistent, even if we shall never know this with utter certainty.

5.1.3 Finitude and the Set of Natural Numbers

As noted, we are assuming the existence of the natural numbers. It will prove very useful then to assume in addition that they jointly form a set; this is not provable from the above axioms. The easiest way to do this is just to add an axiom that declares this explicitly:

NN $\exists s \forall x (x \in s \equiv \text{NUM}(x))$,

i.e., there exists a set s such that for any object x , x is an element of s if and only if x is a natural number. By the axiom of extensionality, there can be only one such set. We will call it \mathcal{N} .

We are now able to define another useful notion. As noted, the structures we will examine will be finite. Nonetheless, it will still be important to be able to *say* explicitly that they are finite, and hence we need to be able to express the concept of finitude. We can do this with the help of the set \mathcal{N} . Specifically,

Def 5: $\text{FINITE}(s) =_{df} \exists n \in \mathcal{N} (s \sim \{m \in \mathcal{N} \mid m < n\})$,

where $t \sim r$ means intuitively that t and r are the same size, i.e., that there is a one-to-one correspondence between them. (This latter notion can also be defined straightforwardly with the set theoretic apparatus at our disposal.) Thus, a set is finite just in case it is the same size as the set that contains all and only the natural numbers less than a given natural number n . The number n is said to be the *cardinality* of the set.

5.1.4 Difference, Intersection, and the Empty Set

Many interesting and important facts about sets are derivable from the above axioms. We will state two. The first is that the existence of the *difference* $a - b$ of two sets a and b , i.e., the set of elements of a that are not in b . ($-$ is thus a new two-place functions symbol.) It is easy to prove that $a - b$ exists: by union, $a \cup b$ exists, and by separation, there is an s that contains just those elements of $a \cup b$ that are both in a and *not* in b .

The next thing we will prove is the existence *intersection* of any two sets, where the intersection of sets a and b is just the set of all objects that a and

b both have as members. We will refer to this set as $a \cap b$, making use of the distinguished two-place function symbol \cap . The proof that $a \cap b$ exists is also easy: by union, $\cup\{a, b\}$ exists; by separation, we then pull out the set of all $x \in \cup\{a, b\}$ such that both $x \in a$ and $x \in b$. In general, we can show that the intersection of any number of sets exists in essentially the same way.

Notice that often there might be no elements common to two sets. Nonetheless, their intersection is a perfectly good set: the empty set. We can prove the existence of the empty set a bit more formally like this. We know there are sets, since first-order logic guarantees the existence of at least one object a , and by pairing it follows that the singleton set $\{a\}$ exists. By the schema of separation, letting φ be the formula $x \not\approx x$ (i.e., $\neg(x \approx x)$), there is a set s that contains all the members x of $\{a\}$ such that $x \not\approx x$, i.e., all the members of $\{a\}$ that are not identical to themselves. But of course there are no members of $\{a\}$ that fit that description. So s is a set with no members, i.e., the empty set. Following the usual practice, we will use the constant \emptyset to refer to this set. Two sets r and s are said to be *disjoint* if they have no members in common, i.e., if $r \cup s = \emptyset$. A set s of sets is said to be *pairwise disjoint* if any two members of s are disjoint.

5.1.5 Functions and Ordered n -tuples

This set theoretic apparatus enables us to provide an elegant account of certain other important notions. First, an extremely versatile and useful notion is that of an *ordered pair*. An ordered pair is similar to a set of two elements, except that unlike a set, which is an unordered collection, there is a first member and a second member. Thus, where $\langle a, b \rangle$ stands for the ordered pair whose first element is a and whose second element is b , what is important about ordered pairs is that they satisfy the following principle:

OP $\forall x \forall y \forall z \forall w (\langle x, y \rangle \approx \langle z, w \rangle \supset (x \approx z \wedge y \approx w))$.

That is, ordered pairs are identical only if their first elements are identical and their second elements are identical, i.e., only if, like any set, they have the same elements, *and*, unlike sets—which have further structure beyond their elements, those elements occur in the same order. The way we write down names for the members of an ordered pair, unlike sets, is therefore

significant, since the first name we write down signifies the first element of the pair, and the second name the second element. For example, whereas $\{a, b\} \approx \{b, a\}$, we have in the case of ordered pairs that $\langle a, b \rangle \neq \langle b, a \rangle$.

As it happens, we need not introduce ordered pairs as a new sort of object, since with a little set theory it is easy to define them as sets of a certain sort. There are many ways to do this, but given that we will have numbers in the semantics for all constraint languages, for our purposes the easiest way to pull this off is simply by "marking" the intended first element of an ordered pair with the number one, and the second with the number two. More precisely, we define the ordered pair $\langle a, b \rangle$ just to be the set $\{\{a, 1\}, \{b, 2\}\}$. It is easy to check that ordered pairs so defined satisfy the above principle. More generally, we can define the notion of an ordered n -tuple in the same way: the n -tuple $\langle a_1, \dots, a_n \rangle$ is defined to be the set $\{\{a_1, 1\}, \dots, \{a_n, n\}\}$.

Given the notion of an ordered n -tuple, we can give a more precise account of the notion of a function. A one-place function f from one set r to another s is just a mapping that takes each element a of r (or some subset of r , if f is partial) to an element $b = f(a)$ of s . Thus, we can simply think of such a function as a set of ordered pairs $\langle a, b \rangle$ where b is the element that a is mapped to by the function f . More generally, an n -place function is a set of ordered $n + 1$ -tuples $\langle a_1, \dots, a_n, a_{n+1} \rangle$ where a_{n+1} is the object that a_1, \dots, a_n are mapped to by the function. Functions thus turn out simply to a type of set. The set of all one-place functions from one set r to another s will be denoted by ${}^r s$.

5.1.6 The Intended Semantics: The Cumulative Hierarchy of Sets

The above gives a good idea of how sets combine and interact, and what sets we can suppose there to be, but it does not provide much of an idea of the intended semantics for set theory and hence for constraint languages generally. The intended picture of the structure of sets within a given domain is known as the *iterative*, or *cumulative*, conception of set. On this conception, sets are hierarchical; they come in *levels*. The lowest level L_0 consists of our initial set of *urelements*, i.e., things that are not themselves sets: numbers, people, machines, buildings, strings, database records, countries, etc. The next level L_1 consists of all possible subsets of L_0 together with the urelements, i.e.,

$L_1 = \text{pow}(L_0) \cup L_0$. The next level L_2 consists of all possible subsets of L_1 together with all the elements L_1 . In general, $L_{n+1} = \text{pow}(L_n) \cup L_n$. Each level is cumulative, i.e., it pulls up the elements of the previous level to join all the sets that could be formed out of those elements. And so it continues through the sequence of natural numbers. The intended semantics for a given constraint language, sets and all, is just the union of all these levels, i.e., $\bigcup_{i \in \mathcal{N}} L_i$.¹⁷

5.2 Constraints Revisited

With the above apparatus in place, we can return to the notion of a constraint and offer an account that is a little more precise. It is our contention that any current information modeling language, and most any language likely to appear on the scene, can be translated into a subset of our language. There is nothing particularly controversial about this claim, given the logical strength of the language we have introduced. The only way to strengthen it in any significant way would be to move to a full higher-order language and logic; but few if any concepts that need to be expressed in the domain of information modeling, database modeling, and the like need anything approaching the power of higher-order logic. Thus, our full-strength first-order language *cum* logic *cum* set theory should be all we need to express anything that can be expressed in any extant or likely modeling language.

The theory here is also expressive enough to define the intended semantic structures that interpret these modeling languages, and expressive enough to define the model theoretic connections—i.e., the interpretations functions and variable assignments—between the languages and those structures. Thus, we will be able to define the notion of *truth* for formulas—or functionally similar syntactic expressions; let us call them *assertions*—of the language, and hence we will be able to characterize when a given semantic structure is a *realization* (in the sense of Section 3.2) of a given set of formulas or, more generally, assertions. We sketch an example of this in the next section.

¹⁷Though this is not anything we can say in the formal constraint language itself, since we can only use it to talk about things *within* its semantic domain—failure to realize this ever-present semantic limitation is in fact what lies behind Russell's paradox.

Using these facts, we can flesh out the notion of a constraint more precisely. Let us call a set of formulas or assertions in a given modeling language a *diagram*. As noted in the introduction, a modeling language might be put to two very different uses: a descriptive, or *de facto*, use, and a prescriptive, or *de jure*, use. Suppose a modeling language ML is being used with respect to a given system S, and the modeler develops a specific diagram D. If ML is being used descriptively, then the system S *as it is* should be capable of being understood as a realization of D. That is, if the diagram D is a correct description of S, it should be possible to consider S abstractly (at the time in question) as a particular instance of an intended semantic structure for ML that makes all the assertions in D true.

On the other hand, if ML is being used prescriptively, then it will not necessarily be possible to consider S as it is to be a realization of D. This will typically be the case for the prescriptive use of ML, since the function of a diagram in such uses is to improve or alter the existing structure of the system in question. The system will fail to realize the diagram. In such a case, the assertions of D must then be considered not as descriptions of S, but as *constraints* on S; they are assertions that *must* be satisfied by any state of S that is to be deemed acceptable. The diagram, that is to say, is prescriptive rather than descriptive. The realizations of D within the intended model theory of ML can thus be thought of as abstract characterizations of the acceptable states of S, the sorts of states that S is permitted to be in.

In both cases, then, *de facto* and *de jure*, D has realizations (so long as it is not contradictory). Only in the former case is it assumed that the current state of the system under scrutiny can itself be considered a realization of D. In the latter, D will in general only have abstract (i.e., set theoretic) realizations which represent the acceptable states of the system. Given this, then, a constraint can be defined simply to be an assertion within a prescriptive diagram.

5.3 Information Structures: An Intuitive Account

Now that we have all this apparatus at our disposal, it should be put to good use. We will demonstrate the power of the apparatus as well as some of the ideas and claims mentioned above by using a constraint language to define a

general type of set theoretic structure suggested by the information modeling technique IDEF1. (An overview of IDEF1 is found in Appendix A.) These structures are similar to the entity-relationship-attribute structures defined by Chen in his seminal 1976 paper [5], though we make explicit the element of *intensionality* in such structures (see below). Despite their relative simplicity, we have found these structures to be very powerful and flexible mathematical tools for characterizing many different types of information-bearing systems. Consequently, for purposes here we will call them *information structures*. In this section we will develop an informal picture of these structures using our apparatus. A more formal treatment is found in Appendix B.

An information structure consists of four different types of objects: *entity classes*, *attribute value classes*, *attributes*, and *links*. Entity classes, attributes, and links are thought of as *intensional* entities, in the sense that, unlike sets, they can have different members, or better, *instances*, across time. Intuitively, the instances of entity classes at any given time are best thought of as featureless "pegs" on which we hang clusters of information. A good model for an instance of an entity class might be an internal pointer within a computer's memory (the featureless entity itself) that points to a collection of records on disk (the clusters of information) associated with, say, a given employee in a company. Since we may keep several different clusters of information on a single real-world individual—for instance, the records on that individual in the role of an employee, and the records on that same individual in the role of a secretary—we think of all the entity classes as disjoint.

Attributes are (intensional) functions from entity class instances to attribute values. Intuitively, an attribute—*SALARY_OF*, for example—takes an instance *e* of an entity class—the class of *employees*, say—to the value of that attribute applied to *e*, viz., in this case the salary of the individual represented by *e*.

In the definition of an information structure one associates with each entity classes a set (possibly empty) of attributes designated to be the ones *owned* by that entity class. For example, the *deparment* entity class might own the attribute *DEPT_NUM_OF*, the *employee* entity class the attributes *EMPLOYEE_NUM_OF* and *WORKS_IN*, and the *secretaries* entity class might own the attribute *TYPING_SPEED_OF*.

Links are functions from entity class instances to entity class instances. That is, a link associates each instance of a given entity class with an instance of another (possibly the same) entity class. Thus, for example, the link *WORKS_IN* maps each instance *e* of the *employees* entity class to the *department* instance that *e* works for. Links come in three flavors: one-to-one, strong many-to-one, and weak many-to-one. To illustrate these, suppose that *E* and *E'* are entity classes in an information structure, and that *l* is a link from *E* to *E'*. Then *l* is one-to-one if no two distinct instances of *E* can possibly be mapped by *l* to the same instance of *E'*. *l* is strong many-to-one if it is not one-to-one and, necessarily, every element of *E'* has at least one instance of *E* mapped to it by *l*. And *l* is weak many-to-one if it is of neither of the above two kinds. Note that if *l* is neither one-to-one nor strong many-to-one, then every instance of *E'* always has zero or more elements of *E* mapped to it by *l*.

Since links are functions, they can often be composed to forge new links between entity classes. Suppose we have a one-to-one link *WORKS_FOR* between the *secretary* entity class and the *employee* entity class to indicate the link between (the cluster of information we keep on) secretaries and (the cluster of information we keep on) the employees they work for. Then by composing this link with the link *WORKS_IN*, we have a new link *WORKS_IN • WORKS_FOR* from *secretary* to *department*, viz., the link that maps the information about a given secretary to the department his or her boss works for.

Since attributes are also functions, we can compose them with links to generate new attributes. For example, if we compose the link *WORKS_IN* with the attribute *DEPT_NUM_OF* that is owned by the entity class *department*, we have a new attribute *DEPT_NUM_OF • WORKS_IN* that maps each employee to the department number of the department he or she works for. The new attribute *DEPT_NUM_OF • WORKS_IN* now associated with *employee* is said to be an *inherited attribute* in *employee*, and we say that *employee inherits* the owned attribute *DEPT_NUM_OF* from the entity class *department* down the link *WORKS_IN*. Finally, we say that the inherited attribute *DEPT_NUM_OF • WORKS_IN* is *derived from* the attribute *DEPT_NUM_OF*.

Certain collections of the attributes—both owned and inherited—associ-

ated with a given entity class are always able to distinguish every member of the class from every other. A collection of attributes that does so in every possible instantiation of the class and which does not contain any unnecessary attributes for that purpose is called a *key class*. Suppose employees in different departments can have the same employee number, but employees in the same department cannot. Then the class consisting of the attributes *DEPT NUM OF • WORKS IN* and *EMPLOYEE NUM OF* constitute a key class for the *employee* entity class. If we were to add *SALARY OF* to this class, it would still perform the same individuating function, but it would not be a key class, since the added attribute is unnecessary to this function.

Since the information we keep about objects, represented in their attribute values, is usually the *only* way to distinguish them, every entity class must have at least one associated key class. In addition, the following conditions are required: (i) if l links the entity classes E and E' , then E inherits from E' all the attributes in some key class of E' down l ; and (ii) if l is a one-to-one link from E to E' , then the inherited attributes of E that are derived from the attributes of E' that E inherits from E' down l themselves form a key class of E . The idea behind (i) is this: suppose that entity class E is linked to E' , and that instance e is mapped to instance e' by this link. Then all the information associated with e' becomes thereby associated with e in virtue of the link between them. The idea behind (ii) is that, if in addition the link is one-to-one, so that no other instance of E besides e is linked to e' , then the information in any key class of E' that distinguishes e' from all other possible instances of E' also must distinguish e from all other possible instances of E .

It will be useful to the reader at this point to see how these informal ideas are explicated formally in the formal framework in the appendix.

6 Summary

The theory we have developed in this paper has several purposes. First, it provides a language for model specification. That is, the theory can be used to provide rigorous definitions of the syntax of a modeling methodology—so

that it is wholly clear exactly what constructs are permissible in the methodology and what are not—and a precise account of its semantics—so that modelers have a clear vision of the sorts of structures they are to be identifying and modeling with the methodology in question.

Second, the theory provides a broad and expressively powerful language that can be used to supplement any given methodology by enabling it to describe and express constraints otherwise inexpressible in the methodology proper. We saw examples of this above. This function of the theory can also be useful in the design or modification phase of a given methodology, in that it can point out clearly the logical form of the sorts of information that one wishes to capture within the methodology.

Finally, the theory is powerful enough to capture the information content of any model within any existing methodology—IDEF1, IDEF1-X, ENALIM,¹⁸ ER,¹⁹ etc.—and also, we believe, any likely model as well. It thus serves as a foundation for the construction of a Neutral Information Representation Scheme which has the capability of capturing information from a model developed using one type of methodology and transferring it—as faithfully as possible—to a model constructed from another type of methodology. We are to the point where we can begin thinking directly about the sorts of algorithms and heuristics that will be needed to carry out such a task. The framework here provides the necessary medium.

¹⁸I.e., Enhanced Natural Language Information Modeling Method.

¹⁹I.e., Entity-Relationship modeling method.

A An Overview of IDEF1

Before attempting any of the other chapters in this report the Integrated Computed Aided Manufacturing (ICAM) DEFinition (IDEF) language, IDEF1 must be understood. IDEF1 has a simple and clean syntax which can be understood quickly. On the other hand, there is an art to modeling in any methodology. IDEF1's design makes it imperative that the modeler understand proper modeling discipline.

As in each of the following chapters, this chapter will begin with a discussion of IDEF1's history and purpose and then move onto its syntax and semantics. Those familiar with the methodologies may not need to read the syntax and semantics sections, but keep in mind that many methodologies have several dialects. In order to understand the metamodels, it is important that the reader understand which dialect is being modeled. In general, the original definitions of methodologies are strictly adhered to.

A.1 History and Purpose

The family of IDEF methodologies is meant to provide methods and languages for discovery, representation, and consensus development of the views of an enterprise necessary to allow for planning and design of integrated information systems. That is, the IDEF methodologies were specifically developed for supporting the domain experts and systems analysts in gathering information about the existing environment and achieving consensus within the environment relative to those descriptions. IDEF0 was developed to model the decisions, actions, and activities within a domain and the relationships among those activities. IDEF1 provides the methods for discovery and representation of the logical structure and relations between basic information groups actually managed by an organization. IDEF2 provides a method for development of quantitative simulation models that allow the study of time varying behavior of a system that is stochastic in nature. IDEF3 supports the direct capture of domain experts descriptions of process flow and object-state transitions. IDEF5 is under development to support the capture and representation of domain knowledge, concepts, and terminology (sometimes referred to as domain ontologies). IDEF1X was the first IDEF methodology

to focus on support of system design activities. IDEF1X data incorporates criteria for efficient conceptual schema design. IDEF4 was developed later to support the design of object-oriented systems, particularly systems encompassing the use of object oriented databases. As a family, the IDEF methodologies provide the modeler with the ability to concentrate on views of an enterprise without using a sledge hammer methodology meant to model all views.

IDEF1 models the information managed within a system, though closely related to IDEF1X it is not a subset of IDEF1X. IDEF1 and IDEF1X are similar, but by providing a methodology for data modeling and consequently conceptual schema database design, the developers of IDEF1X added constructs which cloud the distinction between data which is kept about objects and the objects themselves. This was necessary since a conceptual schema by definition is a type of data dictionary (albeit a complex on-line dictionary used to provide both access and control to distributed electronic heterogeneous databases). Thus, a conceptual schema designer must develop a structure that can both contain the data objects and the information about those data object (such as their physical system location). IDEF1 however, was designed to be both more general and less committed to any particular implementation concept. In a properly developed IDEF1 model there should never be any misconceptions, only the information kept within an organization about objects (physical, abstract or data) is being modeled.

IDEF1 entities need not correspond directly to any particular object in the real world, the IDEF1 model represents the modeler's analysis results. The analysis method results in a reconstruction of the underlying structure and grouping of the information actually managed. In the real world these logical groups of attributes may be distributed over many data artifacts. Also, since data can be kept by the organization about any object, (physical, abstract or data) this flexibility is necessary when attempting to establish information requirements. However, it is not constraining enough when doing database design (hence the need for IDEF1X, IDEF4, Entity Relationship (ER) and other design methods.

As with any of the IDEF methodologies, IDEF1 has primarily been used by defense contractors under contract to the Air Force. Hughes has a proprietary version of IDEF1 called ELKA (Entity Link Key Attribute). IDEF1's

connection with defense projects is good in that a strong underlying analysis method has been developed for the application of IDEF1 modeling. With the emergence of the recognition of the need for a system development framework of methods and the availability of low-cost integrated tools for IDEF1 application, we can expect to see IDEF1 gain more widespread usage.

A.2 Syntax and Informal Semantics

A.2.1 Basic Syntax

The lexicon of the IDEF1 language syntax consists of just four basic symbols (see Figure ??):

- Labeled boxes denoting entity classes,
- Labeled lines with five different types of diamond shaped terminators denoting relation classes,
- Labels inside the boxes denoting attribute classes,
- Parenthesized (or underlined) sets of labels denoting key classes.

A.2.2 Entity Class, Attribute Class, and Key Class

The concept of an entity class is meant to capture the notion of a basic information structure the extension of which at any point in time is a set of informational items called entities. A basic concept behind the notion of an entity is that:

- they are persistent (i.e. the organization expends the resources (time, money, equipment or facilities) to observe, encode, record, organize and store the existence of individual entities),
- they can be individuated (i.e. they can be identified uniquely from other entities).

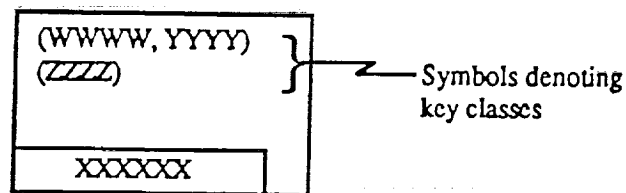
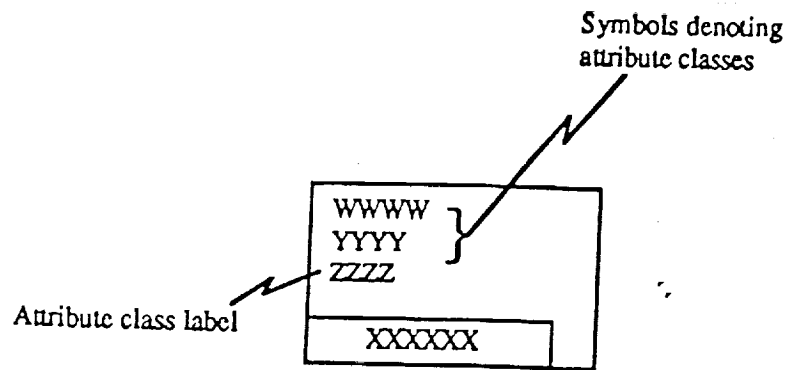
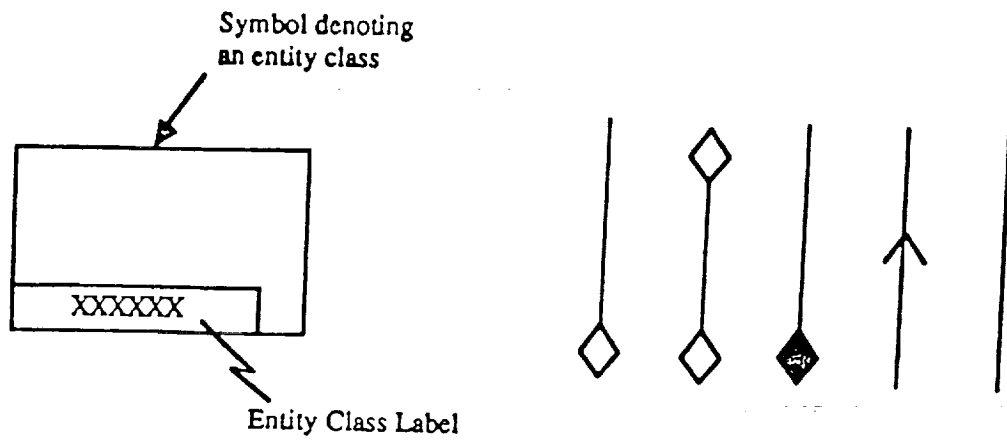


Figure 1: IDEF1 Graphical Lexicon

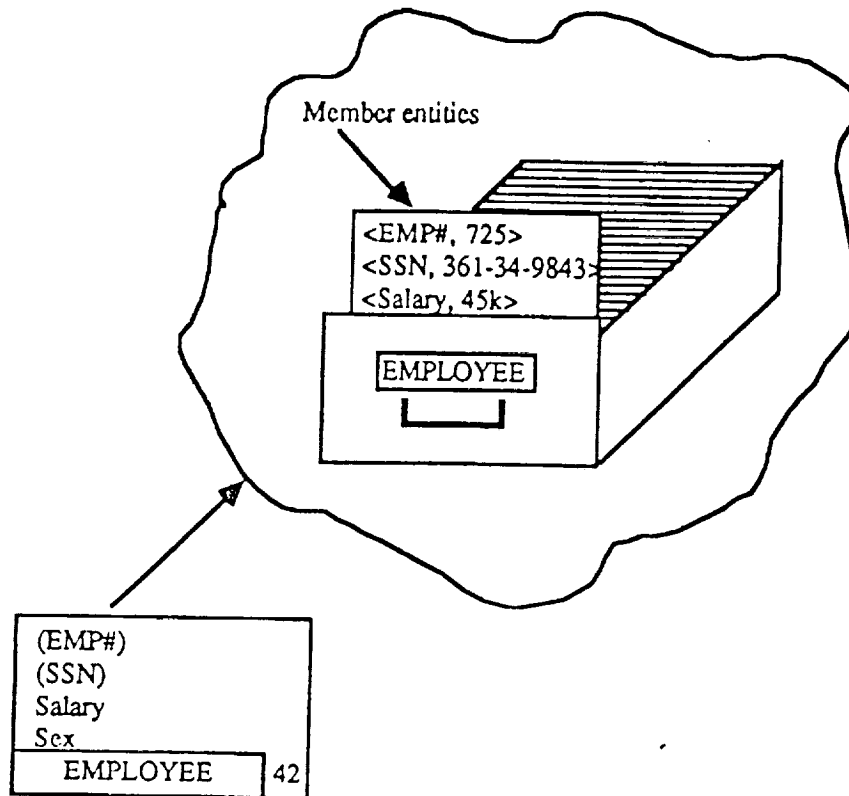


Figure 2: Card file interpretation of an IDEF1 entity class.

The IDEF1 language does not provide a means of representing the individual entities only groups of entities which share exactly the same types of attributes. These groups from an IDEF1 view are called classes. A useful memory aid for this notion is to think of the entity class as a layout for a card file (See Figure ??). An entity class has a name and a unique identification number associated with it, along with a glossary entry and a list of synonyms. An entity class is represented by a rectangular box with the label of the entity class located in the lower left corner of the entity class surrounded by a smaller rectangle and with the entity class number located in the lower right corner of the larger box.

An entity class is actually defined by the set of attribute classes that define the characteristics of all the possible entities in all of its extensions. It is important to note that the set of attributes is more important than the

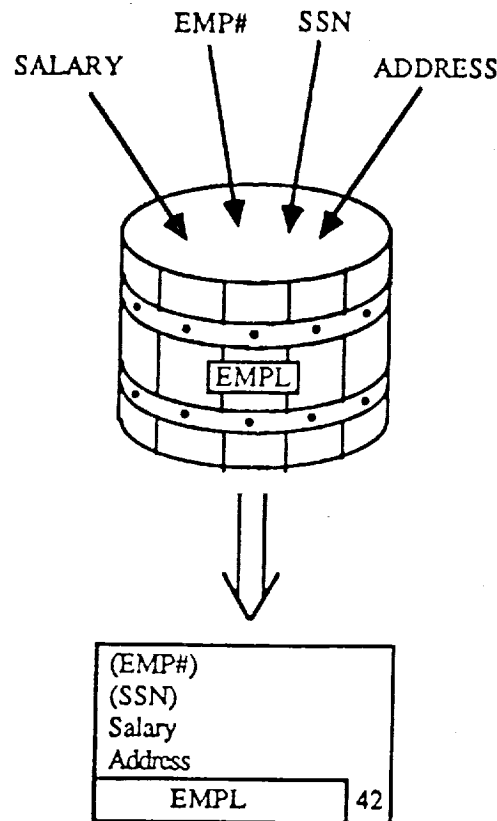


Figure 3: Bucket analogy

notion conveyed by the label on the entity class name! In other words, one can think of the entity class as simply a labeled bucket with no meaning beyond that of the collection of attribute classes it contains (see Figure ??). In fact, it is considered good practice to use an entity class label that does not name a physical or data object in the domain since that could confuse an uninformed reader. The labels of the attribute classes that define an entity class are simply listed in the entity class box below the key class designators and above the entity class label.

The occurrence of the same attribute class in multiple entity class definitions defines a relationship between those entity classes. In order to establish the existence dependency between such entity classes, one entity class must be determined to be the owner of the shared attribute class. Every attribute class that ends up being a part of an IDEF1 model has exactly one owner

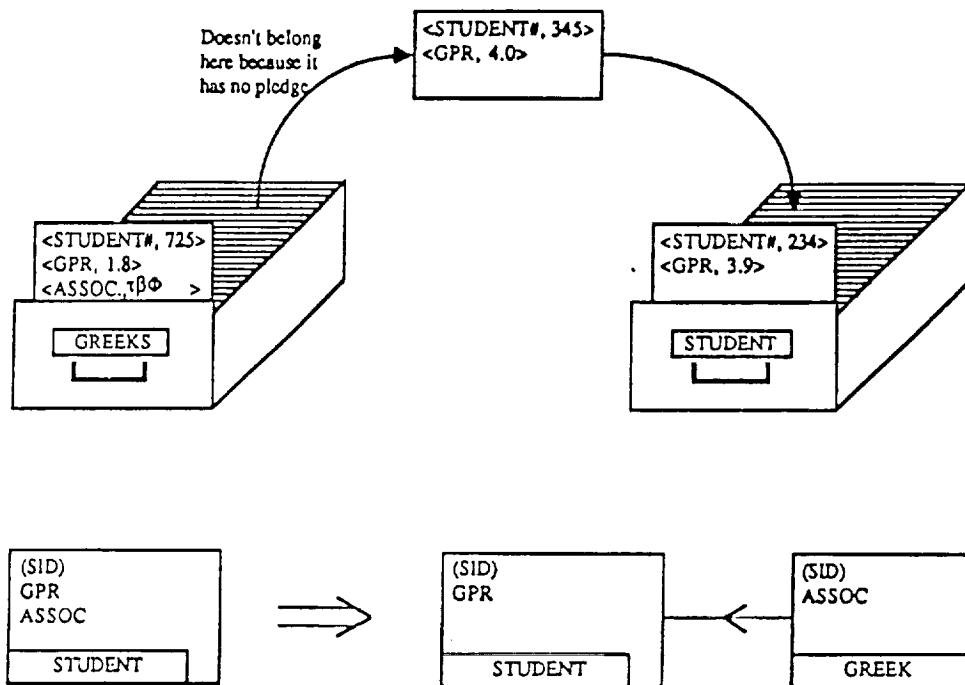


Figure 4: Example of the No-Null rule.

entity class. When deciding on the addition of an attribute class to an entity class, two rules must be followed. The first is referred to as the No-Null Rule. This rule states that no member of an entity class can take a null value for its attribute that corresponds to the added attribute class (Figure ??).

The second rule, the No-Repeat rule, states that no member of an entity class can take more than one value at a time for its attribute that corresponds to the added attribute class (Figure ??).

Each entity class has associated with it at least one key class. A key class is just a special subset of the attribute classes which define the entity class. What makes such key class subsets special is that it can be determined that for any instance, the values of the attributes of that instance (which correspond to the attribute classes in a key class), collectively, will uniquely identify that instance of the entity class from all other instances. In an IDEF1

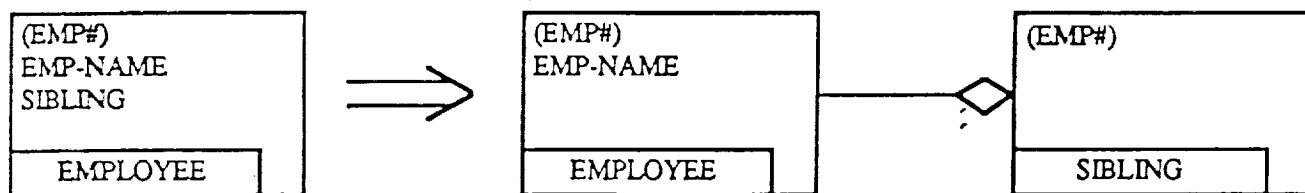
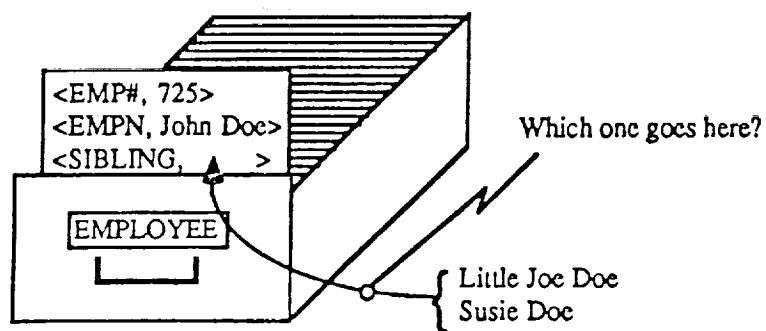


Figure 5: Example of the No-Repeat rule.

diagram, the key class subsets are located in the upper left corner of the entity class for which the key class is being defined. Key classes are not named or labeled, a key class is denoted by enclosing the subset of attribute classes that make up the key class in parentheses or by underlining the subset. In the metamodels of this report we will always use the parenthesis convention. It should be noted that entity classes are allowed to have multiple key classes. The multiple key classes would reflect multiple ways of identifying an entity class instance. For example, in a model of a typical business environment, an instance of an EMPL entity class might have multiple key classes. The first would consist of the employee's name in combination with an employee number. The second key class may consist only of the employee's Social Security Number. In both cases, an EMPL entity class instance could be uniquely identified by either key class (see insert for example).

A.2.3 Link (or Relation) Classes

A link is a binary relationship that exists between two entities established by the sharing of a common attribute(s) which must assume the exact same value in each of the two entities involved in the link. In IDEF1 the generalization of all such links involving instances of the same two classes of entities and the same shared class(es) of attribute(s) is called a *link type*, or (more traditionally) *link class*. A link class establishes a binary relationship between two entity classes that share a common attribute class. A link class is represented by a line running between the boxes of the two entity classes. A label, representing the name of the link class, is displayed over the line representing the link. Because of the attribute class ownership property, a link indicates a dependence of one entity class on the other entity class. The dependent entity class is considered to be existent dependent since a member of that entity class cannot exist unless the corresponding member of the independent entity class already exists. In general IDEF1 uses links to represent common types of organizational constraints (sometimes referred to as business rules) on the information that is managed. It should be noted that not all of the business rules can be represented with the standard IDEF1 language constructs. In another report we describe a constraint language called the Information Systems Constraint Language (ISyCL). ISyCL (pronounced

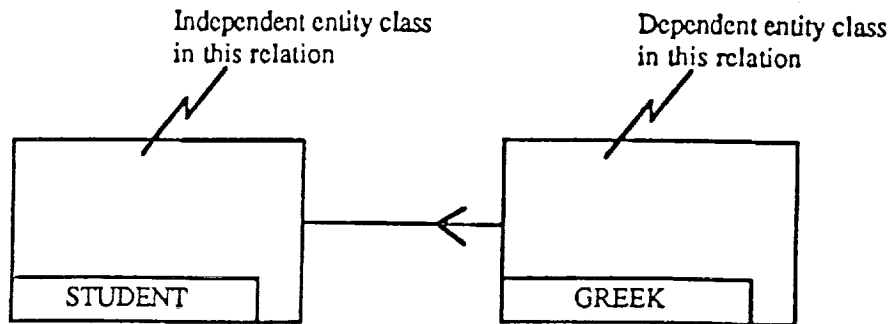


Figure 6: One-to-zero-or-one Link Class.

icicle) is used to augment the standard IDEF1 language as needed in this report to capture some of the more complex rules of individual methods.

A link class also has a cardinality associated with it, specifying the number of members of each entity class that can be involved in a relationship with a single member of the other entity class. Figure ?? shows the syntactic representation of a one-to-zero-or-one (or, thought of functionally in the other direction, one-to-one) relationship.

A link with this cardinality represents the fact that one member of the independent entity class can be associated with zero or one members of the dependent entity class. However, each member of the dependent entity class is associated with one and only one member of the independent entity class.

Figure ?? shows the syntactic representation of a weak one-to-many (or functionally, weak many-to-one) relationship.

In this situation, an independent entity class member can be associated with zero, one, or many dependent entity class members. Again, each member of the dependent entity class is associated with one and only one member of the independent entity class.

Figure ?? shows the syntactic representation of a strong-one-to-many

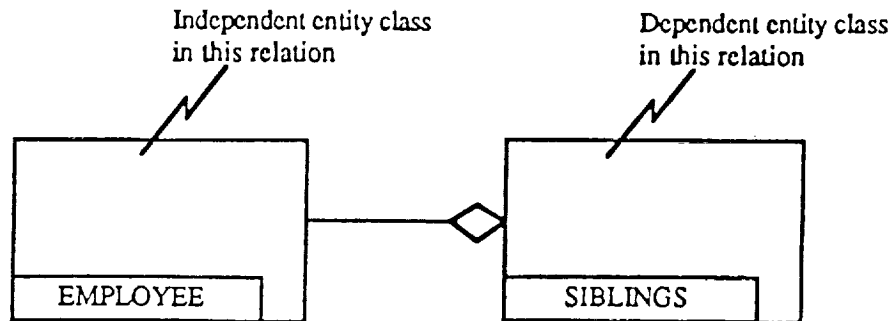


Figure 7: Weak one-to-many Link Class.

(functionally, strong many-to-one) relationship.

Here, the independent entity class member must be associated with at least one instance of the dependent entity class member. Again, each member of the dependent entity class is associated with one and only one member of the independent entity class.

Notice that IDEF1 does not allow a many-to-many relationship or a zero-or-one-to-zero-or-one relationship in what is considered a final model. These relationships make the dependency situation ambiguous. The resolution of such uncertain situations (which often arise in the early phases of the corresponding analysis) often results in the analyst determination that the suspected relationship is unsupported by the analysis data. Alternatively the analyst may discover additional entity class(es) on which both of the entity classes involved in "many-to-many" relationship are independent (an example of this is shown in Figure ??).

Note also that, when specifying a one-to-many link class (either weak or strong), there is no way of constraining that link to a specific upper bound (for example, a one to five relationship). Such details are left to ISyCL if considered absolutely necessary.

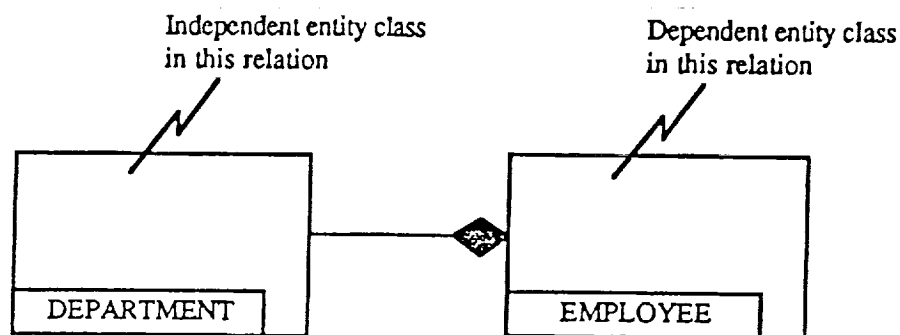


Figure 8: Strong one-to-many Link Class.

A.2.4 Inheritance

Previously we noted that the sharing of attribute classes between two entity classes was the basis for declaring the existence of a link class between those entity classes. However, link classes are generally suspected (or proposed) by the analyst prior to the discovery of exactly which attribute classes are shared. IDEF1 also places certain restrictions on which attribute classes may be (and must be) shared in order for a valid link class to be defined. When a link class is defined between two entity classes, certain information is shared between those entity classes. The attribute classes that make up the key classes of the independent entity class must become attribute classes for the dependent entity class. It is possible for the inherited attribute classes to become part of the key class of the dependent entity class. In fact, the attributes must become part of the key class when a link class has a one-to-zero-or-one link cardinality. In the case of a strong-one-to-many relationship the attributes that are shared cannot make up a key that would be a subset of the key of the independent entity class from which they came.

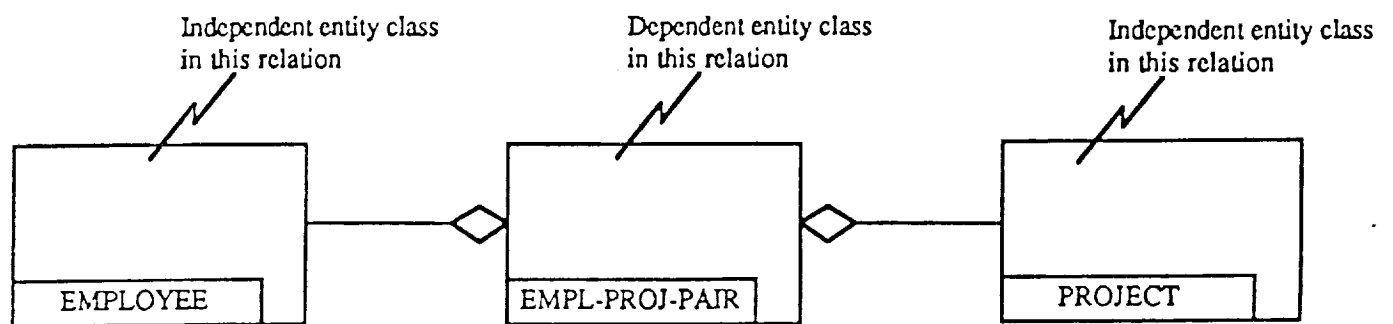


Figure 9: Resolution of a many-to-many relation.

A A Formal Account of Information Structures¹⁹

In this appendix we will make use of our formal apparatus a little more rigorously to give a general definition of an information structure. A note before we continue. Once the notion of a formal language is defined, it is often easier to mingle plain English with the constraint language for easier readability. The only important point is that anything said in this more informal fashion can be stated if need be in a purely formal way. We shall follow this practice here.

In addition to the usual number theoretic and set theoretic apparatus, the elements of our constraint language for the purpose of giving a general definition of information structures will contain a raft of new constants, function symbols, and predicates. These are highlighted below in boldface or italic. Also, since the distinction between object language and metalanguage should be well understood by now, we will revert to the use of the more standard identity predicate = in the object language here.

A.1 Intensional Information Structures.

An *intensional information structure (IIS)* \mathcal{I} is a seven-tuple $\langle E, BL, OA, V, CL, IA, F \rangle$, where

- E is a finite set of objects known as *entity classes*,
- $BL = \bigcup \{BL^{\neg}, BL^{\bullet}, BL^{\diamond}\}$ is the union of three pairwise disjoint finite sets of objects known as *basic link classes* or *basic link types*,
- OA is a finite set of objects known as *owned attributes*,
- V is a set of sets known as *attribute value classes*.
- CL is a finite set of objects known as *composite link classes (types)*, to be described below,
- IA is a finite set of objects known as *inherited attributes*, and

¹⁹This work was partially supported by a grant from Tandem Computer Corporation.

- $F = \{back, front, owner, target, kc\}$ is a set of functions described below.

Intuitively, entity classes are the basic intensional types whose instances can appear in concrete realizations, or instantiations, of an IIS. Basic link types are functions in intension that map the entities of one type E —called the *back* of the link type—to the entities of another (possibly the same) type E' —called the *front* of the link type. And owned attributes are functions in intension that map the entities of a given type E —called the *owner* of the attribute—to a given attribute value class V —called the *target* of the attribute. An attribute value class is thus to be thought of as the range of possible values for a particular owned attribute. Modeling these intuitive connections is the job of the first four functions in F . Specifically,

- $back : BL \longrightarrow E;$
- $front : BL \longrightarrow E;$
- $owner : OA \longrightarrow E;$
- $target : OA \longrightarrow V;$

That is, the function *back* maps a basic link type $l \in BL$ to an entity class $e \in E$, i.e., $e = back(l)$. Similarly for the other functions. To enable us to use more traditional functional terminology, we define the functions $domain = back \cup owner$ and $codomain = front \cup target$.

It is easiest to model the intuitive nature of composite links—the members of CL —as finite sequences (i.e., n -tuples) of basic links. Call any such sequence $s = \langle l_1, \dots, l_n \rangle$ *happy* just in case for all $i < n$ ($i > 0$), $back(l_i) = front(l_{i+1})$.²⁰ Then CL meets the condition

²⁰The idea is that a happy sequence represents a chain of connected link types such that the back of each link type (save the one beginning the chain) is the front of the preceding one in the chain. Now in fact, the actual definition here represents this idea backwards: the intuitive beginning of such a connected chain is actually as defined the last member in the formal representation $\langle l_1, \dots, l_n \rangle$. However, this definition mirrors directly the corresponding IDEF1 syntax for such chains, and hence in the long run makes for a simpler semantics.

C1: $CL \subseteq \{s \mid s \text{ is a happy sequence of basic link types}\}.$

Given this, the functions *back* and *front* can be extended such that,

Def 6: For composite links $L = \langle l_1, \dots, l_n \rangle$,

- $back(L) =_{df} back(l_n);$
- $front(L) =_{df} front(l_1).$

The definitions of *domain* and *codomain* can then be broadened to include these newly defined extensions in the obvious way as well.

Henceforth, let $L = BL \cup CL$. The composite nature of composite link types can be highlighted by defining an operator @ on L such that,

Def 7: For basic links l, l' , and composite links L, L' ,

- $l@l' =_{df} \langle l, l' \rangle$, if $\langle l, l' \rangle \in CL$; otherwise $l@l'$ is undefined;
- $l@L =_{df} \langle l \rangle \frown L$, if $\langle l \rangle \frown L \in CL$; otherwise $l@L$ is undefined;²¹
- $L@l =_{df} L \frown \langle l \rangle$, if $L \frown \langle l \rangle \in CL$; otherwise $L@l$ is undefined;
- $L@L' =_{df} L \frown L'$, if $L \frown L' \in CL$; otherwise $L@L'$ is undefined.

Informally, then, $X@Y$ signifies the composition of the link type X with the link type Y .

Intuitively, an inherited attribute is the composition of a link type with an owned attribute. Thus, modeling composition in terms of sequences as we are, we specify that the set **IA** of inherited attributes meet the condition

C2: $IA \subseteq \{X \mid \text{for some } a \in OA \text{ either for some } l \in BL, X = \langle a, l \rangle, \text{ or for some } L \in CL, X = \langle a \rangle \frown L\}.$

That is, a member of **IA** must be either, in the simplest case, a pair consisting of an owned attribute (i.e., a member of **OA**) and a basic link type (i.e., a

²¹Where $s \frown s'$ is concatenation, i.e., the result of tacking the sequence s' onto the end of the sequence s .

member of **BL**), or else the result of tacking an owned attribute onto the beginning of a composite link type.

We then extend the definition of @ such that,

Def 8: For $a \in \mathbf{OA}$, $l \in \mathbf{BL}$, $L \in \mathbf{CL}$,

- $a@l =_{df} \langle a, l \rangle$, if $\langle a, l \rangle \in \mathbf{IA}$, and undefined otherwise;
- $a@L =_{df} \langle a \rangle \frown L$, if $\langle a \rangle \frown L \in \mathbf{IA}$, and undefined otherwise.

Given this, we have

Def 9: For any inherited attribute $A = a@L$,

- $owned-attr(A) =_{df} a$,
- $link(A) =_{df} L$.

We can then extend the definition of the function *owner* to a function *g-owner* (for “generalized owner”) on the set of all attributes $\mathbf{A} = \mathbf{OA} \cup \mathbf{IA}$ such that,

Def 10: For $a \in \mathbf{OA}$, $g-owner(a) =_{df} owner(a)$; for $A \in \mathbf{IA}$, $g-owner(A) =_{df} back(link(A))$.

That is, the *g-owner* of a given owned or inherited attribute, viewed as a function, is its domain.

The last element *kc* of **F**, is a function from entity classes *e* to sets of subsets of **A**—intuitively, the key classes of *e*—that meets the following conditions:

C3: For all $E \in \mathbf{E}$, $kc(E) \neq \emptyset$,

that is, the set of key classes for any given entity class must be nonempty, i.e., every entity class must have at least one key class.

C4: For all $E \in \mathbf{E}$, and for all $K, K' \in kc(E)$, $K \not\subset K'$.²²

²² \subset , recall, signifies the *proper* subset relation. Note that this condition rules out the possibility of an empty key class, since the empty set is a subset of every set, including itself.

C5: For all $E \in \mathbf{E}$, and for all $A \in \bigcup kc(E)$, $g\text{-owner}(A) = E$,

that is, the attributes in every key class of a given entity class E must be owned by E , i.e., have E as their domain.

Now we define the important notion of a walk and related concepts. These will be used most directly to define information structures.

Def 11: Let $\Sigma = \langle E_1, \dots, E_n \rangle$ be a sequence of entity classes, let $\Lambda = \langle l_1, \dots, l_{n-1} \rangle$ be a sequence of basic link types, and let $W = \langle \Sigma, \Lambda \rangle$. Then

- W is a *walk* (from E_1 to E_n) iff for all $i < n$, $back(l_i) = E_i$ and $front(l_i) = E_{i+1}$, or $back(l_i) = E_{i+1}$ and $front(l_i) = E_i$.
- If W is a walk, then P is *increasing* iff for all $i < n$, if $back(l_i) = E_i$, then $l_i \in \mathbf{BL}^+$, and if $back(l_i) = E_{i+1}$, then $l_i \in \mathbf{BL}^-$.
- W is *cyclic* iff $E_1 = E_n$.
- \mathcal{I} is *connected* iff for all distinct $E, E' \in \mathbf{E}$, there is a walk from E to E' .

A walk, that is, intuitively, is a sequence of entity classes such that each (save the last) E_i is connected to its successor E_{i+1} by a link type l_i , either in one direction or the other. An increasing walk is one such that, when you traverse the link types in a walk from E_1 to E_n , there can be no decrease in cardinality as you move from the extension of one entity class (see paragraph on information structure realizations below) to that of the next. The final two notions are self-explanatory.

Given this apparatus, we can state the last conditions on \mathcal{I} :

C6: \mathcal{I} is connected.

C7: \mathcal{I} contains no increasing cyclic walks.

C8: For all $l \in \mathbf{BL}$, there is some $K \in kc(front(l))$ such that for all $A \in K$, $A@l \in \mathbf{IA}$;²³ if l happens to be 1-1, i.e., if $l \in \mathbf{BL}^+$, then in addition $\{A@l \mid A \in K\} \in kc(g\text{-owner}(A@l))$.

²³I.e., informally, if E is linked to E' via l , then all the attributes A in some key class of E' are inherited into E , i.e., $A@l \in \mathbf{IA}$ so that $g\text{-owner}(A@l) = E$ and $g\text{-owner}(A) = E'$.

C8 captures the conditions on key classes noted in the final paragraph of the previous section.

A.2 Information Structures and their Realizations

A *complete realization* of an IIS \mathcal{I} is a 4-tuple $\langle \mathcal{I}, \mathbf{W}, \mathbf{D}, \text{ext} \rangle$, where \mathbf{W} is a set of indices (intuitively, the set of all possible realizations of \mathcal{I}), \mathbf{D} is a set of objects (intuitively, the set of all possible instances of all the entity classes in \mathbf{E}) and ext is a function that, for each index $w \in \mathbf{W}$, maps elements of $\mathbf{E} \cup \mathbf{OA} \cup \mathbf{BF}$ into objects of the appropriate sort as follows:

C9: For each $E \in \mathbf{E}$, $\text{ext}(w, E) \subseteq \mathbf{D}$.

C10: For all $E, E' \in \mathbf{E}$, and for all $w \in \mathbf{W}$, if $E \neq E'$, then $\text{ext}(w, E) \cap \text{ext}(w, E') = \emptyset$.

C11: For each $A \in \mathbf{OA}$, $\text{ext}(w, A) \in \{f \mid f : \text{ext}(w, \text{owner}(A)) \rightarrow \text{target}(A)\}$.

C12: For each $l \in \mathbf{BL}$,

- if $l \in \mathbf{BL}^-$, then $\text{ext}(w, L) \in \{f \mid f : \text{ext}(w, \text{back}(l)) \xrightarrow{1-1} \text{ext}(w, \text{front}(l))\}$.
- if $l \in \mathbf{BL}^*$, then $\text{ext}(w, L) \in \{f \mid f : \text{ext}(w, \text{back}(l)) \xrightarrow{\text{onto}} \text{ext}(w, \text{front}(l))\}$.²⁴
- if $l \in \mathbf{BL}^\diamond$, then $\text{ext}(w, L) \in \{f \mid f : \text{ext}(w, \text{back}(l)) \rightarrow \text{ext}(w, \text{front}(l))\}$.

Though in any given realization the extension of a member of \mathbf{BL}^- might also be onto, that of a member of \mathbf{BL}^* might also be one-to-one, and that of a member of \mathbf{BL}^\diamond might be either one-to-one or onto, it should not be possible that this could be the case without exception, i.e., in all possible realizations. Thus, as further conditions on an IIS realization we have:

C13: For all $l \in \mathbf{BL}$,

²⁴Where f is *onto* just in case every element of its range has something mapped to it from its domain. The addition function on the natural numbers, for example, is onto (every number is the sum of two numbers—itsself and 0 for instance), while the square function is not (not every number is the square of some number).

- If $l \in \mathbf{BL}^{\rightarrow}$ there is a $w \in W$ such that $\text{ext}(w, l)$ is not onto;
- If $l \in \mathbf{BL}^{\bullet}$ there is a $w \in W$ such that $\text{ext}(w, l)$ is not one-to-one;
- if $l \in \mathbf{BL}^{\diamond}$ there is a $w \in W$ such that $\text{ext}(w, l)$ is not one-to-one and a $w' \in W$ such that $\text{ext}(w', l)$ is not onto.

Note that these conditions cannot be enforced in a database, since, e.g., there is no way to tell whether a one-to-one link which has always been onto will cease being so with the next entry. For instance, by coincidence, it might have always been the case that every employee in a company has one child. Then the extension of the link type from `employee.children` to `employee` has always been one-to-one, despite that fact that this could change as soon as any employee has a second child (supposing this is not prohibited by company policy). The constraints above are thus to be thought of as *design* constraints rather than descriptive constraints, and are important in the construction phase of an information model or database.

For any $L = \langle l_1, \dots, l_n \rangle \in \mathbf{CL}$, $\text{ext}(w, L)$ is the composition of the extensions of the link types l_i at w , i.e., $\text{ext}(w, L) = \text{ext}(w, l_1) \circ \dots \circ \text{ext}(w, l_n)$. Similarly, where A is an inherited attribute $a@L$, $\text{ext}(w, A) = \text{ext}(w, \text{att}(A)) \circ \text{ext}(w, \text{link}(A))$.

As noted above, the role of a key class K within an entity class E is to ensure that in every possible realization of an IIS, the instances of E can be distinguished solely in terms of the values of the (extensions of the) attributes in K in that realization. This is expressed formally by means of the following constraint:

C14: For all $E \in \mathbf{E}$, for all $K \in \text{kc}(E)$, for all $w \in W$, and for all $x, y \in \text{ext}(w, E)$, if $\text{ext}(w, A)(x) = \text{ext}(w, A)(y)$ for all $A \in K$, then $x = y$.

As also noted, key classes must be “minimal” in the sense no proper subset of a given key class may also meet C14; this is expressed as follows:

C15: For all $E \in \mathbf{E}$, for all $K \in \text{kc}(E)$, it is not the case that there is an $S \subset K$ such that for all $w \in W$ and for all $x, y \in \text{ext}(w, E)$, if for all $A \in S$, $\text{ext}(w, A)(x) = \text{ext}(w, A)(y)$, then $x = y$.

Like the conditions on basic links across possible realizations above, and for the same sorts of reasons, C15 also cannot be enforced on a database.

References

- [1] Decker, L. P., and R. J. Mayer, "ISyCL Language Reference," Knowledge Based Systems Laboratory Technical Report No. KBSL-89-1002, Texas A&M University, 1989
- [2] R. Mayer, "Cognitive Skills in Modeling and Simulation," Ph.D. Dissertation, Department of Industrial Engineering, Texas A&M University, 1988
- [3] Ramey, T. *et al.*, "ICAM Information Modeling Manual (IDEF1)," Report No. UM 110231200, Materials Laboratory, AF Wright Aeronautical Laboratories, Wright-Patterson AFB, 1981
- [4] Clocksin, W. F., and C. S. Mellish, *Programming in Prolog*, 3rd edition, New York, Springer Verlag, 1987
- [5] Chen, P., "The Entity-Relationship Model—Toward a Unified View of Data," *ACM Transactions on Database Systems* I (1976), pp. 9-36